ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Refined Techniques for Compression Side-Channel Attacks

Master's Thesis

Yuanming Song

April 14, 2024

Advisors: Prof. Dr. Kenny Paterson, Dr. Lenka Mareková

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

Abstract

A cryptosystem that performs compression before encryption may leak information about the plaintext from the ciphertext length, giving rise to a compression side channel. An attacker with the ability to adaptively embed chosen queries in the plaintext along with a fixed secret can recover the secret from the compression side channel.

Compression side-channel attacks are often assumed to be susceptible to noise and require a large number of interactions. Consequently, some proposed mitigations work by adding noise or limiting interaction, without giving rigorous arguments for their effectiveness.

In this thesis, we designed several refined techniques for compression side-channel attacks from a careful inspection of DEFLATE and zlib. Our techniques enable an attacker to reduce or remove the noise in the compression side channel via amplification, and design complex queries in a modular manner via query programming. We also gave a series of upper bounds on the success probability of compression side-channel attacks.

Acknowledgements

First of all, I wish to thank my supervisors, Lenka Mareková and Kenny Paterson, for spotting the chance for me to write a thesis on such a fascinating topic, for the weekly meetings that sparked many ideas, and for your patience and encouragement over the last six months. Your feedback and advice not only improved this thesis, but also started to influence me in many ways.

I am grateful to Shannon Veitch for helping me find a thesis project.

I thank Kien Tuong Truong for our discussion on the project.

I thank the Delta Chat development team for the coordination and feedback on another project.

Many thanks to my friends for the free food, jokes and company.

Finally, I want to say thank you to my parents. I have no idea what happens next, but I know I would have given up long ago without your support.

Contents

Chapter 1

Introduction

Do not repeat the tactics which have gained you one victory, but let your methods be regulated by the infinite variety of circumstances.

Sun Tzu, *The Art of War*

1.1 Motivation

Encryption and compression are two operations commonly performed on data in transit as well as data at rest. Encryption provides data confidentiality, and compression saves bandwidth or storage by reducing redundancy in data. However, it is now well-known that a cryptosystem that performs compression before encryption may leak information about the plaintext from the ciphertext length, giving rise to a *compression side channel* [29]. Notable compression side-channel attacks include CRIME [48] and BREACH [19], which can recover secrets in the plaintext by exploiting compression in TLS and HTTP.

Intuitively, when compression is performed before encryption, as encryption typically does not hide the length of the compressed plaintext, an attacker can deduce how well the plaintext is compressed from the ciphertext length. While seemingly innocuous, the compression side channel becomes quite useful for adaptive chosen-input attackers, i.e. attackers with the ability to adaptively embed queries of their choice in the plaintext along with a fixed secret.

We sketch the core idea of a compression side-channel attack against the widely-used DEFLATE compression algorithm [10]. Given a prefix to the secret in the plaintext, an attacker can recover the unknown byte following the known prefix by querying all possible combinations of the prefix and candidates for that byte. Because DEFLATE replaces a repeated substring in the input with a reference to its previous occurrence, the plaintext with the correct guess contains a one-byte longer repeated substring and likely has a slightly shorter compressed length. It is thus possible for the attacker to recover the unknown byte by observing the ciphertext lengths and taking the guess that yields the shortest ciphertext. By repeating this process byte-by-byte, the attacker can eventually recover the entire secret.

Given that the compression side channel has been known for more than twenty years and has demonstrated its danger with attacks like CRIME and BREACH, the pervasiveness of compression before encryption in currently deployed systems might come as a surprise. More concerningly, some products continued to perform compression before encryption without any mitigation, even after researchers had demonstrated that these products were vulnerable to compression side-channel attacks [13, 21].

Apart from the performance benefits from using compression, a possible reason for performing compression before encryption is that compression side channels, like other side channels, are colloquially considered to be "noisy" and hard to exploit in practice. Indeed, all existing compression side channel attacks rely on tiny length differences of typically one or two bytes to extract a secret, and most require additional techniques to make the signal more reliable. In practice, compression side-channel attacks may require thousands of queries to extract a short secret, which could be costly and prone to detection.

We think that the compression side channel is not yet well understood, both in theory and in practice. Most of the community's understanding of the compression side channel is closely tied to the CRIME and BREACH attacks, which, by the nature of practical exploits, only needed to explore the details of the DEFLATE compression algorithm to the extent of making the attacks practical on specific targets. For example, Gluck et al. stated in their paper on the BREACH attack that "[a] careful study of DEFLATE will undoubtedly uncover improvements to the attack" [19, Section 4.1]. For some, it might also be tempting to believe that defences that work on significant variants of CRIME and BREACH, such as limiting the number of queries and adding random noise, are able to mitigate all compression side-channel attacks, and a more rigorous analysis is not necessary.

As a motivating example, we point out an implicit assumption on the compression side channel: the compressed lengths of the plaintext co-located with the same query cannot differ by more than a few bytes for different secrets. Consequently, countless efforts were devoted to maintaining as well as suppressing this small signal. This assumption stems from the intuition that similar messages have similar redundancies and therefore should be compressed to similar lengths, which is typically not a design goal of compressors and unsurprisingly does not hold for many compressors [1]. As we will show in Chapter 5, attackers could amplify the apparently small signal to hundreds of bytes or more, easily defeating random noise.

One may further question whether the compression side channel should be considered as a side channel at all. According to Standaert, side-channel cryptanalysis "considers adversaries trying to take advantage of the physical specificities of actual cryptographic devices" [52]. Indeed, most side channels involve physical characteristics related to the implementation of a cryptosystem, such as timing or power consumption, for which there are uncertainties due to environmental factors that perhaps can never be studied exactly. In the compression side channel, however, the information leakage comes solely from the length of the ciphertext, rather than physical attributes. An implication of the comparison is that the compression side channel may be studied and controlled to a very high level of precision. As we will describe in Chapter 6, when equipped with a solid understanding of the underlying compression algorithm, attackers can perform complex queries that were not anticipated in previous works.

1.2 Contributions

This thesis advances the current understanding of the compression side channel in several aspects. Our main contributions are as follows:

1. We present novel techniques to reduce or remove the effect of noise in compression side-channel attacks against DEFLATE and zlib by amplifying differences in compressed lengths with specially crafted queries.
2. We introduce a new paradigm called query programming for designing complex queries in compression side-channel attacks. By exploiting DEFLATE and zlib as state machines, we present several new techniques to design queries that precisely control the information leakage. In particular, our techniques allow an attacker to make fewer number of queries than the divide-and-conquer technique by combining multiple queries.
3. We give a series of upper bounds on the success probability of compression side-channel attacks. We extend the model of compression side-channel attacks by Alawatugoda et al. [2] and combine it with the study on compressor sensitivity by Akagi et al. [1]. We use our extended model to study two simple compressors and padding schemes in terms of compression side-channel attacks.

This thesis also contains several side contributions, which may be of independent interest:

1. We provide an in-depth survey of compression side-channel attacks.
2. Our amplification techniques in Chapter 5 can be seen as giving the first examples on the lack of robustness in DEFLATE/zlib.
3. Our analysis of randomised padding schemes in Section 4.6.2 complement a previous study on randomised padding for length-hiding encryption [9].

1.3 Roadmap

We first review the related work in Chapter 2, and provide the background knowledge for our technical results in Chapter 3.

The main technical results of this thesis are in two parts: We first bound the success probability of compression side-channel attackers in Chapter 4, and then describe our refined attack techniques on DEFLATE/zlib, including amplification in Chapter 5 and query programming in Chapter 6.

We discuss the implications of our results and future work in Chapter 7.

Appendices A and B contain supplementary materials for Chapter 4.

Readers primarily interested in our new techniques can read Sections 3.1, 3.2.1 and 3.3 to 3.5 and Chapters 5 and 6 independently; similarly, readers primarily interested in our theoretical results may find Section 3.2 and Chapter 4 most useful.

Chapter 2

Related work

2.1 Compression and encryption

The interplay between compression and encryption is an intriguing case in cryptography. For a long time, the common belief was that performing compression before encryption could improve security. One frequent argument, first given by Shannon [51], is that compression reduces the redundancy in the plaintext, thereby making it harder to uniquely determine the secret key. Similar statements were reiterated in publications on both theoretical and applied cryptography [5,37,49]. Another argument is that compression can help resist cryptanalysis. As a concrete example, Katz and Schneier [27] described a chosen-ciphertext attack against PGP and several other e-mail encryption protocols, but the attack largely fails to work when the message is first compressed before encryption [23].

However, Kelsey [29] pointed out that the benefits of compression before encryption are largely irrelevant for modern cryptosystems; on the contrary, Kelsey showed that applying compression before encryption may give rise to a compression side channel. We summarise existing compression side-channel attacks in Section 2.2.

Apart from the compression side channel, there are more examples illustrating the danger of performing compression before encryption. Kohno [31] gave a chosen-ciphertext attack against the WinZip encryption scheme by modifying the unauthenticated compression method field. Garman et al. [16] described a gzip format oracle attack against Apple iMessage. Poddebniak et al. [45] showed that while compression significantly complicates their EFAIL attack against OpenPGP, it also allows an attacker to more precisely modify and exfiltrate the plaintext. Recently, Schwarzl et al. [50] demonstrated several attacks that recover encrypted data via information leakage from decompression time differences. Note that we classify the attacks of Schwarzl et al. as timing or decompression side-channel attacks rather than compression side-channel attacks, because their attacks exploit the timing side channels in decompressor implementations.

Despite being exploited in many attacks, there is still no consensus on whether to perform compression before encryption, a choice that arguably depends on the threat model and cost. For example, Gellert et al. [17] suggested that performing compression before encryption can help mitigate fingerprinting attacks, and argued that "the *general* advice to disable compression, independent of a given attacker model, can be misleading and harmful to security" [17, Appendix A].

Finally, we note that there are studies on the counter-intuitive construction of compressing encrypted data without using the secret key [15, 24, 30], which is mainly of theoretical interest and is out of the scope of this thesis.

2.2 Compression side-channel attacks

Kelsey [29] was the first to describe several compression side-channel attacks that can detect or extract secret strings from the plaintext. However, practical compression side-channel attacks on deployed systems came much later.

2.2.1 The "CRIME family"

CRIME, developed by Rizzo and Duong [48], is a compression side-channel attack that recovers session cookies from HTTP requests by exploiting compression in TLS and SPDY. Rizzo and Duong described several techniques to make the attack work reliably and reduce the number of queries. They also hypothesized that similar techniques could be used to extract secrets from HTTP responses, such as personal information and CSRF tokens. We note that Adam Langley was the first to sketch a possible compression side-channel attack against SPDY and HTTPS to extract secret cookies.[1]

The CRIME attack required a Machine-in-the-Middle (MitM) attacker that can directly observe the sizes of HTTP requests from the target's device to the server. The **TIME** [4] attack by Be'ery and Shulman changed the target to HTTP responses, where compression is widely used and cannot be simply disabled without incurring notable performance damage. TIME uses timing differences to infer changes in HTTP payload sizes, removing the need for a MitM attacker. **HEIST** [56] shares the same idea as the TIME attack, but Vanhoef and Van Goethem made several practical improvements, including a way to measure the exact HTTP payload sizes from timing differences. They also demonstrated the HEIST attack for HTTP/2.

The **BREACH** [19] attack by Gluck et al. also targeted compressed HTTPS responses as TIME did, but it requires attackers to directly observe the sizes

[1] https://groups.google.com/g/spdy-dev/c/B_ulCnBjSug/m/rcU-SIFtTKoJ

of HTTP responses. Gluck et al. described several techniques to reduce noise and improve stability, and suggested some possible mitigations to the attack.

Karakostas and Zindros [26] built **Rupture**, a framework for performing compression side-channel attacks. They employed statistical methods to reduce noise and bypass block alignment, and proposed several optimizations, including divide-and-conquer and browser parallelization. We remark that Gluck et al. have suggested that statistical methods can be used to mitigate noise [19, Section 3.1], and Thomas Pornin was likely the first to introduce the divide-and-conquer method for improving the compression side-channel attack.[2]

Note that all attacks in this section target compressors that implement the DEFLATE algorithm [10], which we will introduce in Section 3.3. Readers interested in the relations between these attacks can find more details in [54].

2.2.2 Recent developments

Compression side-channel attacks have recently expanded to a more diverse set of targets, including databases and encrypted messaging applications.

Hogan et al. [21] introduced DBREACH attacks against encrypted databases. They argued that existing techniques were ineffective in the database setting, and proposed a series of new techniques to exploit the compression side channel in encrypted databases, such as aligning guesses with page boundaries and computing a heuristic compressibility score. DBREACH uses repeated queries to amplify the noisy signal in order to extract secrets from the database. In addition to DEFLATE, DBREACH also works on LZ4 and Snappy, two compressors based on the LZ77 algorithm [60].

Paterson et al. [41] discovered a compression side-channel attack against Threema, a Swiss encrypted messaging application. They showed that an attacker with physical access to the victim's device could extract their private key in the DEFLATE-compressed and encrypted backup by repeatedly triggering backups with attacker-controlled payloads.

Fábrega et al. [13] described a compression side-channel attack against the encrypted backup mechanism of WhatsApp, which enables an attacker to recover a message from a known set of messages in the zlib-compressed and encrypted backup by performing a binary search. They demonstrated the attack for small sets of messages.

[2] https://security.stackexchange.com/questions/19911/crime-how-to-beat-the-beast-successor/19914#19914

2.2.3 Traffic analysis

There is also a line of research on leveraging compression side channels for traffic analysis of protocols, such as TLS [44] and encrypted VoIP [57, 58]. More recently, Albrecht et al. [3] discovered a compression side channel in the Bridgefy SDK, and demonstrated a compression side-channel attack that recovers broadcast messages from a set of known plaintexts.

Traffic analysis is not the focus of this thesis, as we primarily study "CRIME-style" attacks, in which an active attacker tries to extract a secret in the plaintext by performing adaptive queries to a compressed length oracle.

2.3 Mitigations

A successful compression side-channel attack appears to be contingent on several factors, including (i) co-location of the secret and attacker-controlled data in the plaintext, (ii) the use of an "exploitable" compressor, (iii) the ability to observe or infer the lengths of the compressed data, and (iv) a large amount of interaction. Therefore, if disabling compression altogether proves to be too costly, then one might deploy mitigations that target one or more of the above factors. However, many proposed defences have been shown to be either ineffective or impractical, and few have received a rigorous evaluation.

In this section, we briefly review some mitigations for compression side-channel attacks. Our primary focus is on mitigations for compression side-channel attacks in general, and readers interested in deploying mitigations for some specific applications should also consult other sources, e.g. [47].

2.3.1 Separating secrets

Separating secrets from user inputs or not compressing secrets can mitigate or potentially eliminate the compression side channel [2,19,25,36,42]. However, this approach either requires developers to manually mark or move secret fields [25], which may be impractical [19], or requires a mechanism to automatically identify secrets [2,36,42], which can be error-prone and might introduce additional side channels if implemented badly.

2.3.2 Switching compressors

Since all existing compression side-channel attacks target LZ77-based compressors, switching to other compressors might help mitigate the attacks. However, as Kelsey [29] remarked, many other compressors can be attacked using similar techniques, and "the same side channels clearly exist and can be exploited" [29, Section 8] for the Burrows-Wheeler transform [6], which is used in bzip2.

Compressors that only adapt to the input to a small extent are likely less vulnerable to compression side-channel attacks. Alawatugoda et al. [2] described a compressor with a fixed dictionary and provided a security proof. However, this compressor performs relatively poorly, and the proof is flawed (see Section 4.5.1). Zieliński [59] designed SafeDeflate, a variant of DEFLATE with reduced information leakage, and gave a security analysis. However, SafeDeflate also suffers from poor performance, especially if the alphabet for secrets is large.

HPACK [43] and QPACK [33] are compression algorithms for HTTP headers in HTTP/2 and HTTP/3, respectively. They are designed to mitigate compression side-channel attacks on HTTP headers like CRIME [48], using fixed Huffman coding and restricted dynamic dictionaries. Note that HPACK and QPACK are not general-purpose compressors, but tailored for HTTP header compression.

2.3.3 Masking secrets

Gluck et al. [19] described a mitigation that they attributed to Tom Berson, in which the server masks a secret S with a one-time pad P and sends $P \| P \oplus S$ to the client. Intuitively, an attacker cannot effectively accumulate information about a secret across multiple queries if the secret is masked. However, this method has several drawbacks. First, masking secrets increases the length of the compressed data, especially if secrets are non-random [19] or many secrets are present [26]; second, developers have to identify secrets in advance; finally, although intuitively secure, masking secrets has not received a formal analysis in the context of compression side-channel attacks.

2.3.4 Adding noise

Noise can be introduced to the input, the internal state or the output of a compressor, making the compression side channel harder to exploit [19, 29]. An attacker may need to perform more queries to average out noise in observed ciphertext lengths, which can be costly and prone to detection. However, adding noise can only slow down but not completely eliminate compression side-channel attacks [19,26].

Degabriele [9] studied random padding schemes in the context of length-hiding encryption. Degabriele showed that Gaussian padding is better than uniform padding when multiple samples are available, and demonstrated the superiority of Gaussian padding on a variant of the CRIME attack.

Palacios et al. [38] proposed HTB, a mitigation against the BREACH attack which works by modifying the gzip library to add a random fake filename for each compression. Palacios et al. showed with a simplified model and experiments that HTB can significantly increase the number of queries an attacker needs to perform. HTB is recommended on the website of the BREACH attack [46].

2.3.5 Limiting interaction

Rate limiting and monitoring can slow down the BREACH attack [19]; they can likely also mitigate other compression side-channel attacks, which often require a large number of adaptive queries.

Some proposed mitigations also make it more difficult for attackers to perform queries. For BREACH, such mitigations include CSRF protection [19], performing referrer checks [47] and disabling third-party cookies [26].

2.3.6 Combining compression with encryption

Kelsey suggested preprocessing the input by applying a stream cipher that "generates a keystream of extremely low Hamming weight" [29], which might be regarded as an encrypt-then-compress-then-encrypt construction.

It might be tempting to consider combining compression with encryption in non-trivial ways to achieve better security. However, as Kelley and Tamassia [28] observed, a (perfectly correct) cryptosystem cannot be both be IND-CPA secure and achieve meaningful compression. We point out that the squeeze cipher by Kelley and Tamassia [28], which combines compression and encryption, was only shown to satisfy a weak security notion that is already satisfied by the compress-then-encrypt construction, and therefore does not count as a mitigation.

2.4 Security models

There are numerous works on formally modelling side-channel attacks. For example, Köpf and Basin [32] developed an information-theoretic model for adaptive side-channel attacks, and applied the model to timing and power side channels. These models could potentially be used to study compression side-channel attacks, but no such connections were explicitly made to our knowledge.

Kelley and Tamassia [28] studied the security of combining compression and encryption. They defined a new security notion called entropy-related IND-CPA (ER-CPA) security, which is similar to IND-CPA security, but requires that the challenges come from a fixed set of messages. Their work considered exclusively sets of messages that compress to the same length, on which a compress-then-encrypt construction can already be ER-CPA secure. Therefore, as Alawatugoda et al. [2] commented, the work of Kelley and Tamassia does not capture compression side-channel attacks.

Alawatugoda et al. [2] defined several new security notions for compression side-channel attacks on secret cookies, including cookie recovery (CR), random cookie indistinguishability (RCI) and chosen cookie indistinguishability (CCI). Here CR is a weak security notion sufficient for "CRIME-style" attacks, RCI is a stronger indistinguishability-style notion, and CCI, being the strongest of the three notions, means that it is infeasible for attackers to distinguish between two cookies of their choice embedded in the plaintext. Alawatugoda et al. used these security notions to formally analyse the security of two mitigations. They showed that separating secrets from user inputs achieves CCI security, and their fixed-dictionary fixed-width compression scheme achieves CR security.

Security models for length-hiding encryption [9, 17, 40, 55] are related to modelling compression side-channel attacks. We highlight two previous works that discussed compression side-channel attacks in the context of length-hiding encryption.

Gellert et al. [17] provided a new methodology for quantifying the effects of length-hiding encryption, and analysed the effect of padding, compression and mode of operation on fingerprinting attacks using small-scale real-world datasets. However, their security model does not capture "CRIME-style" attacks and only concerns fingerprinting adversaries.

Degabriele [9] revisited the previous work [40, 55] and proposed a unified security model for length-hiding encryption. Degabriele used the new security model to demonstrate the security of the pad-then-encrypt composition; more precisely, Degabriele reduced the length-hiding security of a pad-then-encrypt construction to the channel simulatability of the symmetric encryption scheme and the difference hiding (D-HIDE) security of the padding length distribution, where D-HIDE characterises the difficulty of distinguishing between the original distribution and a shifted distribution with multiple samples for information-theoretic adversaries.

2.5 Robustness of compressors

Our theoretical bounds in Chapter 4 and amplification techniques in Chapter 5 are related to studies on the robustness of compressors to small changes in their inputs.

The term "one-bit catastrophe" refers to the phenomenon that changing a single bit or character in the input of some compressors could incur a drastic change in the length of the compressed data. Motivated by earlier works such as [35], Lagarde and Perifel [34] studied the one-bit catastrophe for the LZ78 algorithm, and Giuliani et al. [18] studied this phenomenon for the Burrows-Wheeler transform.

Akagi et al. [1] generalised the one-bit catastrophe and introduced the notion of compressor sensitivity, a compressor property that measures the maximum change in the length of compressed data with regard to the substitution, insertion or deletion of a single character. They derived bounds on the sensitivities of various theoretical compressors.

Chapter 3

Background

3.1 Terminology

We list some important terminology that we use throughout the thesis. Readers are encouraged to check it to avoid possible misinterpretations of our work, especially since compressors and compressed data formats are used interchangeably in many sources, which may be inaccurate in the context of compression side-channel attacks.

- *Compression:* In this thesis, compression refers specifically to lossless[1] data compression; see also Definition 3.5.
- *Compression side channel:* The leakage of information about the plaintext from the length of the compressed data. Our study does not cover timing side channels in decompressors [50].
- *DEFLATE*: We refer to DEFLATE as a general compression algorithm that conforms to the algorithm details in the DEFLATE compressed data format specification [10, Section 4].
- *LZ77*: A variant of the LZ77 algorithm [60] described in the DEFLATE specification [10, Section 4].
- *zlib*: A library that implements DEFLATE compressors and decompressors.[2] Note its difference to the zlib data format [12].
- *GNU Gzip*: A software tool for data compression using DEFLATE.[3]
- *gzip*: A compressed data format [11]. Both zlib and GNU Gzip, among others, can be used to generate compressed data in the gzip format.

[1]We only use the lossless property for proofs in Appendix A.
[2]`https://zlib.net/`
[3]`https://www.gnu.org/software/gzip/`

3.2 Definitions

3.2.1 Notation

Let $\mathbb{Z}$ be the set of all integers, $\mathbb{N} = \{0, 1, 2, \ldots\}$, and $\mathbb{N}_+ = \mathbb{N} \backslash \{0\} = \{1, 2, \ldots\}$. For $k \in \mathbb{N}$, let $[k] = \{1, 2, \ldots, k\}$. We use $\log x$ to denote the binary logarithm of x, and use $\ln x$ to denote the natural logarithm of x. We define $0 \ln 0 = 0$ for simplicity.

An *alphabet* is a finite set Σ, where the size of the alphabet is $|\Sigma|$. A *string* s is an element of the set Σ^*. Let o denote the empty string. Let $\Sigma^+ = \Sigma^* \backslash \{o\}$. Let $|s|$ denote the *length* of a string s, and the length of the empty string is 0. Let $\perp$ be a specified error symbol that is distinct from all strings.

For a string s of length $\ell \in \mathbb{N}_+$, and for each $i \in [\ell]$, let $s[i]$ be the i-th character of s; we can write s as $s[1]s[2] \ldots s[\ell]$, or vice versa. For a string s of length $\ell \in \mathbb{N}_+$, and for each $i, j \in [\ell]$, where $i < j$, let $s[i, j] = s[i]s[i+1] \ldots s[j]$.

For strings a and b, let $a \| b$ denote their concatenation. For strings a and b where $|a| = |b|$, let $\mathsf{HD}(a, b)$ denote the *Hamming distance* between a and b, which is the number of indices at which a and b differ; that is, $\mathsf{HD}(a, b) := |\{i \in [\ell] : a[i] \neq b[i]\}|$.

For a probability distribution $\mathcal{D}$ on a countable sample space S, which we will refer to in this thesis as a *distribution* in short, let $\Pr_{\mathcal{D}}[\cdot]$ be the *probability function* of $\mathcal{D}$, which is a function that maps every subset of S to a value in $[0, 1]$, and let $\Pr_{\mathcal{D}}[s]$ be a shorthand for $\Pr_{\mathcal{D}}[\{s\}]$ for every $s \in S$; let $\mathrm{Supp}(\mathcal{D})$ be the *support* of $\mathcal{D}$, i.e. $\mathrm{Supp}(\mathcal{D}) := \{s \in S : \Pr_{\mathcal{D}}[s] \neq 0\}$; let $x \leftarrow\!\!\$\; \mathcal{D}$ denote sampling x according to $\mathcal{D}$.

We only consider discrete random variables in this thesis. For a random variable $\mathbf{x}$, let $\mathrm{Supp}(\mathbf{x})$ be the support of its corresponding distribution.

For a finite set S, let $x \leftarrow\!\!\$\; S$ denote sampling x uniformly at random from S. For a set S, let $\mathbf{1}_S$ be an *indicator function* for S on domain $X \supseteq S$, i.e. $\forall x \in X$,

$$\mathbf{1}_S(x) := \begin{cases} 1 & \text{if } x \in S, \\ 0 & \text{if } x \notin S. \end{cases}$$

For a randomised algorithm $\mathcal{A}$, let $\mathcal{A}(\ldots)$ represent the distribution of outputs when running $\mathcal{A}$ with the specified inputs, and let $\mathcal{A}(\ldots; r)$ denote running $\mathcal{A}$ with fixed random coins r.

We write `<L`x`,D`y`>` for a back-reference of length x and distance y in LZ77, where we may replace y with `*` for an unspecified distance.

3.2.2 Distance measures

Definition 3.1 (statistical distance) *For two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ on a sample space S, the statistical distance between $\mathcal{D}_1$ and $\mathcal{D}_2$ is*

$$\mathsf{SD}(\mathcal{D}_1, \mathcal{D}_2) = \sum_{s \in S} \max\left\{0, \Pr_{\mathcal{D}_1}[s] - \Pr_{\mathcal{D}_2}[s]\right\}.$$

For two random variables $\mathbf{x}$ and $\mathbf{y}$ with an image S, their statistical distance is the statistical distance between their corresponding distributions, i.e.

$$\mathsf{SD}(\mathbf{x}, \mathbf{y}) = \sum_{s \in S} \max\left\{0, \Pr[\mathbf{x} = s] - \Pr[\mathbf{y} = s]\right\}.$$

We list some well-known properties of the statistical distance.

Lemma 3.2 *For two random variables $\mathbf{x}$ and $\mathbf{y}$ with an image S,*

- *(equivalent definition)* $\mathsf{SD}(\mathbf{x}, \mathbf{y}) = \frac{1}{2} \sum_{s \in S} |\Pr[\mathbf{x} = s] - \Pr[\mathbf{y} = s]|$,
- *(range)* $0 \leq \mathsf{SD}(\mathbf{x}, \mathbf{y}) \leq 1$,
- *(symmetry)* $\mathsf{SD}(\mathbf{x}, \mathbf{y}) = \mathsf{SD}(\mathbf{y}, \mathbf{x})$,
- *(triangle inequality) for every random variable $\mathbf{z}$ of image S, we have* $\mathsf{SD}(\mathbf{x}, \mathbf{z}) \leq \mathsf{SD}(\mathbf{x}, \mathbf{z}) + \mathsf{SD}(\mathbf{z}, \mathbf{y})$,
- *(data-processing inequality) for every function f on the domain S, we have* $\mathsf{SD}(f(\mathbf{x}), f(\mathbf{y})) \leq \mathsf{SD}(\mathbf{x}, \mathbf{y})$.

Definition 3.3 (KL divergence) *For two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ on a sample space S, where* $\operatorname{Supp}(\mathcal{D}_1) \subseteq \operatorname{Supp}(\mathcal{D}_2)$, *the Kullback–Leibler (KL) divergence between $\mathcal{D}_1$ and $\mathcal{D}_2$ is*

$$\mathsf{KL}(\mathcal{D}_1, \mathcal{D}_2) = \sum_{s \in \operatorname{Supp}(\mathcal{D}_1)} \Pr_{\mathcal{D}_1}[s] \ln\left(\frac{\Pr_{\mathcal{D}_1}[s]}{\Pr_{\mathcal{D}_2}[s]}\right).$$

The KL divergence between two random variables $\mathbf{x}$ and $\mathbf{y}$ with an image S where $\operatorname{Supp}(\mathbf{x}) \subseteq \operatorname{Supp}(\mathbf{y})$ *is defined analogously as*

$$\mathsf{KL}(\mathbf{x}, \mathbf{y}) = \sum_{s \in \operatorname{Supp}(\mathbf{x})} \Pr[\mathbf{x} = s] \ln\left(\frac{\Pr[\mathbf{x} = s]}{\Pr[\mathbf{y} = s]}\right).$$

Lemma 3.4 (Pinsker's inequality) *Let S be a countable set, $\mathbf{x}$ and $\mathbf{y}$ be two random variables with the image S, such that* $\operatorname{Supp}(\mathbf{y}) \subseteq \operatorname{Supp}(\mathbf{x})$. *Then*

$$(\mathsf{SD}(\mathbf{x}, \mathbf{y}))^2 \leq \frac{1}{2} \mathsf{KL}(\mathbf{x}, \mathbf{y}).$$

3.2.3 Compression and encryption

We give the definition of a compression scheme adapted from Alawatugoda et al. [2] and a standard definition for a symmetric-key encryption scheme.

Definition 3.5 (Compression scheme) *A compression scheme on input alphabet Σ and output alphabet Ω is a pair (C, D), where*

- *C is a randomised algorithm called a compressor on Σ, which takes $m \in \Sigma^+$ as input and returns $z \in \Omega^+$, and*
- *D is a deterministic algorithm called a decompressor on Ω, which takes $z \in \Omega^+$ as input and returns $m' \in \Sigma^+ \cup \{\perp\}$,*

such that

$$\forall m \in \Sigma^+, \forall z \in \mathsf{Supp}(C(m)), D(z) = m.$$

Remark 3.6 *We highlight a difference from [2, Definition 2]: our definition is on fixed input Σ^+ and output Ω^+ rather than their subsets, in order to simplify the discussion on sensitivity. Real-world compressors that have an upper bound on the input length can be naturally augmented to satisfy our definition.*

Remark 3.7 *We note that most compressors are deterministic, but our definition allows us to capture random noise in compression; see examples in Section 2.3.4.*

Definition 3.8 (Symmetric-key encryption) *A symmetric-key encryption scheme over key space $\mathcal{K}$, message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ is a tuple $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$, where*

- *KGen is a randomised algorithm that takes no input and returns $k \in \mathcal{K}$,*
- *Enc is a randomised algorithm that takes $(k, m) \in \mathcal{K} \times \mathcal{M}$ as input and returns $c \in \mathcal{C}$, and*
- *Dec is a deterministic algorithm that takes $(k, c) \in \mathcal{K} \times \mathcal{C}$ as input and returns $m' \in \mathcal{M} \cup \{\perp\}$,*

such that

$$\forall k \in \mathcal{K}, \forall m \in \mathcal{M}, \forall c \in \mathsf{Supp}(\mathsf{Enc}(k, m)), \mathsf{Dec}(k, c) = m.$$

We now restate the composition of compression and encryption described in [2], modified to suit our definitions. We also fixed a small error in [2, Definition 3], where decryption may pass $\perp$ to the decompressor.

Definition 3.9 (Compress-then-encrypt [2, Definition 3]) *Let $\Gamma = (C, D)$ be a compression scheme on input alphabet Σ and output alphabet Ω. Let $\Pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key encryption scheme over key space $\mathcal{K}$, message space $\mathcal{W}$ and ciphertext space $\mathcal{C}$, where $\mathcal{W} \subseteq \Omega^+$. Let $\mathcal{M} \subseteq \Sigma^+$ be such that $\forall m \in \mathcal{M}$, $\mathsf{Supp}(C(m)) \subseteq \mathcal{W}$.*

The compress-then-encrypt construction from Γ and Π yields a symmetric-key encryption scheme $\Pi' = (\mathsf{KGen}', \mathsf{Enc}', \mathsf{Dec}')$ over key space $\mathcal{K}$, message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$, where

- $\mathsf{KGen}' := \mathsf{KGen}$,
- $\forall k \in \mathcal{K}, \forall m \in \mathcal{M}$, $\mathsf{Enc}'(k, m) := \mathsf{Enc}(k, C(m))$, *and*
- *For* $(k, c) \in \mathcal{K} \times \mathcal{C}$, $\mathsf{Dec}'(k, c)$ *is defined in Fig. 3.1.*

```
Dec'(k, c)
1 : $w \leftarrow \mathsf{Dec}(k, c)$
2 : if $w = \perp$ then
3 :     return $\perp$
4 : $m \leftarrow D(w)$
5 : if $m \notin \mathcal{M}$ then
6 :     return $\perp$
7 : return $m$
```

Figure 3.1: Decrypt-then-decompress, the decryption process for the compress-then-encrypt construction. We stress that this is a simplified definition, and implementations that directly follow this definition may be vulnerable to format oracle attacks [16] or side-channel attacks [50].

3.2.4 IND-CPA

We give a standard definition of the IND-CPA security game for a symmetric-key encryption scheme $\Pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ with the Left-or-Right (LoR) oracle in Fig. 3.2. The advantage of $\mathcal{A}$ is

```
Game IND-CPA($\mathcal{A}, \Pi$)
1 : $b \leftarrow\$ \{0, 1\}$
2 : $b' \leftarrow\$ \mathcal{A}^{\mathsf{LoR}}()$
3 : return $b = b'$

Oracle LoR($m_0, m_1$)
1 : if $|m_0| \neq |m_1|$ then
2 :     return $\perp$
3 : return $\mathsf{Enc}(k, m_b)$
```

Figure 3.2: The IND-CPA security game for symmetric-key encryption scheme $\Pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$; we require that $m_0, m_1 \in \mathcal{M}$ for queries to the LoR oracle.

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi}(\mathcal{A}) := |2 \Pr[\mathsf{IND\text{-}CPA}(\mathcal{A}, \Pi) \Rightarrow \mathsf{true}] - 1|.$$

Readers who wish to interpret our work in the context of Alawatugoda et al. [2] should be aware that they used a single-query definition for IND-CPA.

3.2.5 Cookie recovery

We consider a variant of the cookie recovery (CR) game defined by Alawatugoda et al. [2], which captures compression side-channel attacks where an attacker tries to recover a secret cookie by observing the encrypted cookie compressed together with some attacker-chosen data. The main differences from Alawatugoda et al. [2] are:

1. We force the cookies to be of the same length. In [2], the lengths of cookies could be different, which is one of the reasons why their reduction from random cookie indistinguishability (RCI) to CR [2, Theorem 7] is incorrect.[4]

2. We assign a probability distribution $\mathcal{D}_{\mathcal{CK}}$ to secret cookies. For example, the "cookies" may be personal information like credit card numbers or phone numbers in real life. More precisely, we consider a cookie distribution over $\mathcal{CK}$, where $\mathcal{CK} \subseteq \Sigma^k$ for some $k \in \mathbb{N}_+$.

3. The oracle query now has a length restriction of $L - k$. This is because (i) attackers may only control a small input field in practice, and (ii) it is possible to reduce the number of queries by making longer queries, as in the example of binary search and what we will show in Chapter 5.

```
Game CR($\mathcal{A}, \Pi, \mathcal{D}_{\mathcal{CK}}, L$)
1: $k \leftarrow\!\!\$\ \mathsf{KGen}()$
2: $ck \leftarrow\!\!\$\ \mathcal{D}_{\mathcal{CK}}$
3: $ck' \leftarrow\!\!\$\ \mathcal{A}^{E_1, E_2}()$
4: return $ck = ck'$
```

```
Oracle $E_1(m', m'')$
1: $m \leftarrow m' \| ck \| m''$
2: if $|m| > L$ then
3:     return $\perp$
4: $c \leftarrow\!\!\$\ \mathsf{Enc}(k, m)$
5: return $c$
```

```
Oracle $E_2(m)$
1: $c \leftarrow\!\!\$\ \mathsf{Enc}(k, m)$
2: return $c$
```

Figure 3.3: A variant of the cookie recovery (CR) security game [2, Figure 3] for symmetric-key encryption scheme $\Pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ on message space $\mathcal{M}$, cookie distribution $\mathcal{D}_{\mathcal{CK}}$ on $\mathcal{CK} \subseteq \Sigma^k$, and maximum length L, where Σ is an alphabet, $k, L \in \mathbb{N}_+$, $k \leq L$, and $\bigcup_{i=k}^{L} \Sigma^i \subseteq \mathcal{M}$.

Figure 3.3 shows our variant of the cookie recovery (CR) game [2]. The advantage of $\mathcal{A}$ is

$$\mathsf{Adv}^{\mathsf{CR}}_{\Pi, \mathcal{D}_{\mathcal{CK}}, L}(\mathcal{A}) := \Pr[\mathsf{CR}(\mathcal{A}, \Pi, \mathcal{D}_{\mathcal{CK}}, L) \Rightarrow \mathsf{true}].$$

[4]Another error in their proof is that their bound did not account for random guessing. More specifically, they stated that for every CR adversary $\mathcal{A}$, there exists a RCI adversary $\mathcal{B}^{\mathcal{A}}$ such that $\mathsf{Adv}^{\mathsf{CR}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{RCI}}(\mathcal{B}^{\mathcal{A}})$, but a correct bound should be $\mathsf{Adv}^{\mathsf{CR}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{RCI}}(\mathcal{B}^{\mathcal{A}}) + 1/|\mathcal{CK}|$.

3.2.6 Multisample distinguisher

Degabriele [9] studied the best advantage for an unbounded adversary to distinguish between two distributions with multiple independent samples available, in order to evaluate the effectiveness of different random padding schemes in terms of length hiding. Theorem 3.10 gives a slightly modified version of Degabriele's multisample distinguisher theorem [9, Theorem 4.3].

Theorem 3.10 (corollary of [9, Theorem 4.3]) *Let M and N be two distributions over a sample space Ω. Let $S = \mathrm{Supp}(M) \cap \mathrm{Supp}(N)$, and assume $S \neq \emptyset$. For $q \in \mathbb{N}_+$, let $\mathbf{x}_1, \dots, \mathbf{x}_q$ be independent random variables distributed according to M, and let $\mathbf{y}_1, \dots, \mathbf{y}_q$ be independent random variables distributed according to N. Then*

$$\mathsf{SD}((\mathbf{x}_i)_{i\in[q]}, (\mathbf{y}_i)_{i\in[q]}) \leq q \cdot \sum_{x\in\Omega\setminus S} (\Pr_M[x] + \Pr_N[x]) + \sqrt{\frac{q}{2}\mathsf{KL}(\widetilde{M}, \widetilde{N})},$$

where $\widetilde{M}$ and $\widetilde{N}$ are two distributions defined on the sample space S such that

$$\Pr_{\widetilde{M}}[X] := \frac{\Pr_M[X]}{\Pr_M[S]} \quad \text{and} \quad \Pr_{\widetilde{N}}[X] := \frac{\Pr_N[X]}{\Pr_N[S]}$$

for every $X \subseteq S$.

As Degabriele observed, the bound in Theorem 3.10 consists of two terms that grow with different rates as the number of samples q increases; therefore, padding schemes that perform well when $q = 1$ may not still be a good choice when q is large, such as uniform padding.

Chen et al. also derived a similar result on the relation between statistical distance and KL divergence [7, Lemma 1], which may be used to obtain a variant of Theorem 3.10. More generally, Theorem 3.10 is related to studies on information-theoretic indistinguishability, e.g. [8,39].

3.3 DEFLATE

DEFLATE [10] is a widely used lossless compressed data format. Since our focus is on compressors that conform to the algorithm details described in the DEFLATE format specification [10, Section 4], such as zlib and GNU Gzip, we will refer to DEFLATE as a general compression algorithm from now on. The exact details of the DEFLATE algorithm may vary for different implementations and parameter choices, and readers can refer to our description of zlib in Section 3.4 for a concrete DEFLATE implementation.

Note that, for historical reasons as well as for performance, there are compressors that deviate from the DEFLATE algorithm but can still produce compressed data in the DEFLATE format, such as Zopfli.[5] These compressors are out of the scope of this thesis.

Very roughly speaking, the DEFLATE algorithm works as follows. First, it runs a variant of the LZ77 algorithm [60] (Section 3.3.1) to scan the input data for repeated strings, for each repetition creating a back-reference to its last occurrence. Then, it uses Huffman coding [22] (Section 3.3.2) to further compress the LZ77 output.

Our description of DEFLATE does not cover its full details; we refer interested readers to the DEFLATE specification [10] and an explanation of DEFLATE by Feldspar [14].

3.3.1 LZ77

LZ77 [60] is a compressor that works by replacing repeated strings in the input data with their back-references. Many variants of the LZ77 algorithm exist, and in this thesis, we refer to LZ77 as the variant described in the DEFLATE specification [10, Section 4].[6]

Note that the LZ77 algorithm in DEFLATE works on byte strings; in other words, the input alphabet for LZ77 is $\Sigma = \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$.

Basic idea

The LZ77 algorithm operates in a streaming manner. It scans the input data with a pointer for repetitions, keeping a sliding window of window_size up to 32KiB for bytes before the pointer.

At each new pointer position, LZ77 tries to find the longest substring in the sliding window that matches one of the substrings starting from the pointer. If there are multiple matches of the same length in the sliding window, then LZ77 prefers the one that is nearest to the pointer. LZ77 ignores any match that is shorter than the minimum length of min_match $= 3$ bytes or longer than the maximum length of max_match $= 258$ bytes.

If LZ77 finds a match in the sliding window, then the algorithm outputs a back-reference, which encodes the length of the matched substring and the distance to its previous occurrence in the sliding window, and advances the pointer past the matched substring. Otherwise, the algorithm outputs the byte at the pointer position and advances the pointer by one byte.

For example, running LZ77 on input

```
draw drain train
```

[5] https://github.com/google/zopfli

[6] Some may find this LZ77 variant closer to the LZSS algorithm [53].

gives us

```
draw <L3,D5>in t<L4,D6>
```

where we write <Lx,Dy> for a back-reference of length x bytes and distance y bytes. Note that the substring `rain` in `train` can still match the substring `rain` in `drain`, even though `drain` has already been replaced by `<L3,D5>in`.

Algorithm 1 shows a simplified version of the LZ77 algorithm described above, where we omit some details like the hash table and lazy matching, and we assume the algorithm always searches the whole sliding window for the longest match.

Algorithm 1: A simplified version of the LZ77 algorithm.

```
Input: A string $s = s_1 s_2 \ldots s_n$.
Output: A sequence of bytes and back-references $(t_1, t_2, \ldots, t_m)$.
$p \leftarrow 1$;
$m \leftarrow 0$;
while $p < n$ do
    $m \leftarrow m + 1$;
    $(l, d) \leftarrow (0, 0)$;
    for $j \leftarrow 1$ to window_size do
        $l_j \leftarrow \max\{k : k \leq \text{max\_match} \wedge s_{p-j} \ldots s_{p-j+k-1} = s_p \ldots s_{p+k-1}\}$;
        if $l_j \geq l$ then
            $(l, d) \leftarrow (l_j, j)$;
    if $l \geq$ min_match then
        $t_m \leftarrow$ <L$l$,D$d$>;
        $p \leftarrow p + l$;
    else
        $t_m \leftarrow s_p$;
        $p \leftarrow p + 1$;
return $(t_1, t_2, \ldots, t_m)$;
```

Hash table

The LZ77 algorithm maintains a chained hash table for the positions of all 3-byte substrings in the sliding window to help find repetitions. Substrings are put into the hash table in the order of their positions. LZ77 inserts each substring position before the head of its corresponding chain in the hash table, and sets the current substring position as the new head. As a result, substring positions on every hash chain are ordered by their proximity to the LZ77 pointer. The oldest entry may be purged when adding a new substring to the hash table.

To find the longest match, LZ77 computes the hash digest of the 3-byte substring starting from the pointer, and traverses the chain in the hash table that corresponds to the hash digest. For each entry on the chain that stores a position corresponding to the same 3-byte substring, LZ77 looks for the longest repetition starting from that position in the input. This process stops after the corresponding hash chain has been traversed, or if sufficiently many positions have been checked.

For better speed, LZ77 may be configured not to insert substrings in (long) matches to the hash table, at the cost of a generally worse compression ratio. For example, suppose we configure LZ77 to not add new substrings in any match to the hash table. Then, running LZ77 on input

```
draw drain train
```

yields the output

```
draw <L3,D5>in train
```

instead of

```
draw <L3,D5>in t<L4,D6>
```

in the usual case. The reason is that LZ77 does not add positions of substrings `rai` and `ain` in `drain` to the hash table since their positions fall within a match `dra`, and therefore LZ77 cannot find any match in the hash table when scanning `train`.

Lazy matching

To achieve a better compression ratio, LZ77 may also be configured to perform an additional operation called lazy matching after a match has been found, where LZ77 tries to find a longer match starting from the byte just after the pointer. If such a match is found, LZ77 discards the current match and takes the longer new match. More specifically, it outputs the current byte and advances the pointer by one byte, starting lazy matching again on the new match. Otherwise, LZ77 proceeds as usual; that is, LZ77 outputs the back-reference for the current match and advances the pointer past it.

For example, running LZ77 on input

```
train draw drain
```

gives us

```
train draw <L3,D5>in
```

without lazy matching, and

```
train draw d<L4,D11>
```

with lazy matching. This is because, after finding a repetition `dra` in `drain`, LZ77 searches for a longer match starting from `r` in `drain`, and finds `rain` in `train`. Since `rain` is longer than `dra`, LZ77 takes `rain` and discards the original match.

3.3.2 Huffman coding

Huffman coding [22] is a well-known method for data compression. Huffman coding generates prefix-free codes called Huffman codes according to the frequencies of the symbols, assigning shorter codes for symbols that occur more frequently, to minimise the expected code length. Conceptually, Huffman coding works by constructing a binary tree called a Huffman tree, in which each leaf node represents a symbol, and each path from the root to a leaf node represents a Huffman code.

The Huffman coding process in DEFLATE is rather complicated, and it is often regarded as one of the main sources of noise in the compression side channel (e.g. [19]). Here we only highlight some parts of Huffman coding that are relevant to our attacks in Chapters 5 and 6.

DEFLATE uses two sets of Huffman codes to encode the LZ77 output, with one set of Huffman codes for both literals and match lengths, and another set of Huffman codes for match distances. They must meet certain requirements such that one can define the Huffman codes by a sequence of the codes' lengths for each symbol ordered according to the alphabet. The lengths of the Huffman codes are measured in bits.

In LZ77, match lengths take values from 3 to 258, and match distances take values from 1 to 32,768. Lengths and distances are respectively grouped and coded in LZ77: lengths take codes from 257 to 285, and distances take codes from 0 to 29. To differentiate between different lengths or distances with the same code, DEFLATE appends zero or more extra bits to each length or distance code. The exact number of extra bits depends on the code: for example, a distance of 100 bytes is encoded as bits `110100100` in the LZ77 output, a concatenation of the code $1101_{(2)} = 13$ for distances 97–128 and five extra bits $00100_{(2)} = 4$. Note that the extra bits do not participate in Huffman coding and therefore stay unchanged in the final output.

Table 3.1 shows the codes and extra bits for distance encoding in DEFLATE. Note that longer distances generally require more extra bits, the rationale being that closer repetitions tend to occur more frequently. The length encoding in DEFLATE follows a similar logic, generally requiring more extra bits to encode longer lengths; we refer interested readers to [10, Section 3.2.5].

A DEFLATE implementation may choose whether to use fixed or dynamic Huffman codes based on input data by comparing which option yields a

Code	Extra Bits	Distance	Code	Extra Bits	Distance
0	0	1	15	6	193–256
1	0	2	16	7	257–384
2	0	3	17	7	385–512
3	0	4	18	8	513–768
4	1	5–6	19	8	769–1,024
5	1	7–8	20	9	1,025–1,536
6	2	9–12	21	9	1,537–2,048
7	2	13–16	22	10	2,049–3,072
8	3	17–24	23	10	3,073–4,096
9	3	25–32	24	11	4,097–6,144
10	4	33–48	25	11	6,145–8,192
11	4	49–64	26	12	8,193–12,288
12	5	65–96	27	12	12,289–16,384
13	5	97–128	28	13	16,385–24,576
14	6	129–192	29	13	24,577–32,768

Table 3.1: Codes and extra bits for distance encoding in DEFLATE. [10, Section 3.2.5]

better compression ratio. If dynamic Huffman codes are used, then the final output also includes the length sequences of the Huffman codes, compressed first with run-length encoding and then with another set of Huffman codes. As with most existing attacks on compression,[7] our attack techniques work for both fixed and dynamic Huffman codes.

3.4 Zlib

For a concrete implementation of DEFLATE, we look at zlib,[8] a compression library that implements DEFLATE compressors and decompressors. Zlib is widely used to produce compressed data in zlib, DEFLATE or gzip format. Our description of zlib is based on zlib version 1.3,[9] but it likely also applies to many other zlib versions, as the zlib compressor implementations have been relatively stable. We remark that GNU Gzip implementation of DEFLATE is very similar, albeit not identical, to the zlib implementation.

3.4.1 LZ77 in zlib

The LZ77 implementation in zlib closely follows our description of the LZ77 algorithm in Section 3.3.1, with some adjustable parameters to control the compression speed and quality.

[7] In fact, some attacks may work more reliably when the Huffman codes are fixed, and EFAIL [45] actually requires the Huffman codes to be fixed, albeit in a different context.

[8] https://zlib.net/

[9] https://github.com/madler/zlib/releases/tag/v1.3

By default, zlib uses a sliding window of around 32KiB. It is also possible to increase min_match (default 3) or decrease max_match (default 258) in zlib. The hash table in LZ77 contains the positions of all substrings of min_match bytes in the sliding window. The zlib implementation of LZ77 uses an unkeyed "rolling hash" whose output ranges from 0 to $2^{\mathsf{hash\_bits}} - 1$, where hash_bits is 15 by default. In this thesis, we focus on the hash function with regard to the default parameters min_match $= 3$ and hash_bits $= 15$; other cases should work similarly. The hash value of a 3-byte string *abc* is

$$\mathsf{hash}(a, b, c) := \left(a \cdot 2^{10} + b \cdot 2^{5} + c\right) \mod 2^{15},$$

where each byte is treated as an unsigned integer.

At each new position, LZ77 in zlib checks at most max_chain (default 128) entries in the hash table for a match; the search stops once the longest match found reaches at least nice_length (default 128) bytes.

Zlib defines two compression modes, deflate_fast and deflate_slow. In general, the former has better speed, while the latter produces shorter output. The default compression mode is deflate_slow. In addition, zlib has a deflate_stored mode that does not perform compression and merely stores the input data in an appropriate format.

In deflate_fast, LZ77 does not perform lazy matching, and does not insert new substrings into the hash table for matches strictly longer than max_insert bytes, with the exception that the first substring of min_match bytes in any match is always inserted into the hash table when trying to find repetitions.

In deflate_slow, LZ77 adds all substrings in the sliding window to the hash table. Moreover, LZ77 performs lazy matching when the current match is strictly shorter than max_lazy (default 16) bytes. In addition, LZ77 considers a match of at least good_length (default 8) bytes as "good enough", for which it only checks at most max_chain/4 entries in the hash table to find a better match in lazy matching.

We note that LZ77 in zlib cannot create back-references to a string that starts at the first byte of the input data. To quote a relevant comment in the zlib source code: "To simplify the code, we prevent matches with the string of window index 0."[10] This fact is generally irrelevant to compression side-channel attacks in practice, but might be good to know for readers who wish to test our programming techniques in Chapter 6.

[10] https://github.com/madler/zlib/blob/09155eaa2f9270dc4ed1fa13e2b4b2613e6e4851/deflate.c#L1340C46-L1341C61

3.4.2 Compression levels

Zlib defines 10 compression levels, represented by integers from 0 to 9. The default compression level in zlib is 6. Generally speaking, DEFLATE in zlib runs at a lower speed but produces shorter compressed data at higher compression levels, and vice versa.

Table 3.2 shows the configuration for each compression level in zlib.[11] Zlib does not perform compression at level 0. For levels 1 to 3, zlib uses the deflate_fast compression mode, which does not insert substrings in a match above max_insert bytes into the hash table; for levels 4 to 9, zlib uses the deflate_slow compression mode, which performs lazy matching if the current match is below max_lazy bytes.

Level	good_length	max_lazy	max_insert	nice_length	max_chain	Mode
0	-	-	-	-	-	stored
1	-	-	4	8	4	fast
2	-	-	5	16	8	fast
3	-	-	6	32	32	fast
4	4	4	-	16	16	slow
5	8	16	-	32	32	slow
6	**8**	**16**	**-**	**128**	**128**	**slow**
7	8	32	-	128	256	slow
8	32	128	-	258	1024	slow
9	32	258	-	258	4096	slow

Table 3.2: Configuration for each compression level in zlib. We use "-" to represent parameters that are not used at the corresponding compression level, and we mark the default level (6) in zlib and its corresponding configurations in bold. We use "stored", "fast" and "slow" as abbreviations of deflate_stored, deflate_fast and deflate_slow, respectively.

3.5 Existing techniques

In this section, we briefly review several techniques for compression side-channel attacks. Our focus is on attacks that recover a secret in the plaintext, i.e. cookie recovery (CR) attackers (Section 3.2.5), and we abstract away the technical details of the targeted protocols and applications.

3.5.1 Kelsey's secret extraction attacks

Kelsey [29] proposed two compression side-channel attacks for extracting a secret in the plaintext. In Kelsey's attack model, an attacker can choose N prefixes $P_0, \ldots, P_{N-1}$ for the secret S and learn the compressed length of

[11]Table reproduced from the zlib source code; see https://github.com/madler/zlib/blob/09155eaa2f9270dc4ed1fa13e2b4b2613e6e4851/deflate.c#L112C1-L124C67

$P_i \| S$ for each $i \in \{0, \ldots, N-1\}$; the goal of the attacker is to extract the secret S.

Adaptive chosen input attack [29, Section 6.1]

An adaptive attacker can set a prefix P such that

$$P \| S = \mathsf{prefix} \| \mathsf{guess} \| \mathsf{filler} \| \mathsf{prefix} \| S,$$

where prefix and filler are two strings that have little in common, both known not to occur in S, and guess is a guess for the first k characters of S.

As Kelsey explained, if the guess is correct, then $P \| S$ contains a longer repeated substring ($\mathsf{prefix} \| \mathsf{guess}$) and its compressed length should be smaller than in the case where the guess is incorrect. After finding the correct guess, the attacker can modify prefix and move on to guess the next k characters of S, and continue this process until all characters in S are recovered.

If we assume that S is chosen uniformly at random from Ω^ℓ, where $\ell \in \mathbb{N}_+$ is a multiple of k, then the average number of chosen prefixes is in theory $(\ell |\Omega|^k)/(2k)$. However, Kelsey implemented the attack against zlib and observed that the attack was quite unstable, often requiring backtracking [29].

Non-adaptive chosen input attack [29, Section 6.2]

A non-adaptive attacker can set $P_0, \ldots, P_{N-1}$ to be all strings of a certain length k, and estimate the likelihood of P_i appearing in S from the compressed length of $P_i \| S$; the attacker can recover the secret by piecing together prefixes that are most likely to appear in S.

If S is sampled from Ω^ℓ, then the attacker needs to choose $|\Omega|^k$ prefixes. In Kelsey's implementation, the attacker did not uniquely determine S, but rather provided a small list of candidates for S.

3.5.2 CRIME/BREACH

In CRIME [48] and BREACH [19], an attacker tries to recover a secret cookie ck from a plaintext, i.e. $\ldots \| Q \| \ldots \| P \| ck \| \ldots$ or $\ldots \| P \| ck \| \ldots \| Q \| \ldots$, where P is a known prefix (e.g. `secret=`), and ck is a cookie value chosen uniformly at random from Ω^ℓ. The attacker can adaptively choose Q, a part of the plaintext located either before or after the prefix and the secret cookie, and observe the length of the DEFLATE-compressed and encrypted ciphertext.

Generally, both CRIME and BREACH attacks recover the secret cookie ck character-by-character, similar to Kelsey's adaptive chosen-input attack [29]. Both attacks employed new techniques to improve stability and reduce the number of queries.

Two Tries

Rizzo and Duong designed the Two Tries method for the CRIME attack, which was later adapted by Gluck et al. for the BREACH attack.

We sketch the general idea of the Two Tries method. In order to recover the first unknown character in ck, an attacker iterates over the cookie alphabet Ω, and makes two specifically crafted queries Q_1 and Q_2 for each guess g, with the query Q_2 being a permutation of Q_1. If the resulting ciphertexts of the two queries are of different lengths, then the attacker regards g as the correct guess and proceeds to recover the next unknown character of the cookie in a similar way, until the entire cookie is recovered.

In theory, a Two Tries attacker makes $\ell|\Omega|$ queries on average.

Central to the Two Tries method is the design of Q_1 and Q_2, to which CRIME and BREACH take slightly different approaches.

Two Tries in CRIME. The Two Tries method in CRIME makes two queries

$$Q_1 = \mathsf{filler}\|P\|g \quad \text{and} \quad Q_2 = P\|g\|\mathsf{filler}$$

for a guess g. Here the queries are located before the prefix and the secret cookie in the plaintext, and filler is a string such that the substring $P\|g$ in Q_2 is outside of the sliding window when the LZ77 pointer reaches the cookie $P\|ck$.

According to Rizzo and Duong [48], if the guess g is the correct, then LZ77 matches $P\|g$ in Q_1 but not Q_2, causing the compressed lengths of the two plaintexts to differ; otherwise, if the guess g is incorrect, then LZ77 cannot find any match between the cookie and the queries, so the compressed lengths of the two plaintexts should be the same.

However, it appears to us that the attacker needs to use a small prefix P for the queries, preferably of length $\mathsf{min\_match} - 1$, as otherwise LZ77 can still match P in Q_1 but not in Q_2 if the guess is incorrect.

Two Tries in BREACH. The BREACH [19] attack used a variant of the Two Tries method, in which an attacker makes two queries

$$Q_1 = P\|g\|\mathsf{padding} \quad \text{and} \quad Q_2 = P\|\mathsf{padding}\|g$$

for each guess g. Here the queries may be before or after the prefix and the secret cookie in the plaintext, and padding is a short string that does not have any character in Ω.

If the guess g is correct, then Q_1 contains a longer repeated substring $P\|g$ and likely yields a shorter ciphertext; otherwise, if the guess is incorrect, then both Q_1 an Q_2 likely contains only the repeated substring P, so their corresponding plaintexts are likely to be compressed to the same length.

The BREACH attack employed several techniques to improve the stability of Two Tries by mitigating the effect of Huffman coding, including *guess swap*, *charset pools*, *dynamic padding* and *looking ahead* [19, Section 2.4]. However, the attack may still miss the correct guess or identify multiple candidates for an unknown character.

16K-1

Rizzo and Duong [48] described an attack technique called 16K-1, which can reduce the average number of queries by an attacker to around $\ell\lceil\log|\Omega|\rceil$. However, the attack leverages a feature in TLS and does not apply to other compression side channels in general.

To recover the first unknown character in *ck*, a 16K-1 attacker maintains a candidate set cands for the unknown character, initialised to be Ω, and repeats the following process until $|\mathsf{cands}| = 1$:

- first, the attacker makes a query Q of around 16KiB such that the plaintext is split into two TLS records, each compressed separately, and the only unknown character in the first record is $ck[1]$;
- then, the attacker checks every $g \in \mathsf{cands}$, only keeping a guess in cands if substituting the unknown character in the record to g yields a ciphertext of the same length.

After that, the attacker regards the only element in cands as the unknown character in *ck*, and proceeds to guess the rest of *ck* character-by-character in similar ways.

It is clear that the technique can recover *ck* correctly with high probability. Rizzo and Duong [48] did not specify the method of selecting the query Q, but it can be inferred that the Q consists of a half of the guesses in cands, so that a half of the guesses are encoded more efficiently by Huffman coding, and and the other half are encoded with more bytes. Therefore, each query can shrink the size of $|\mathsf{cands}|$ by 2, equivalent to a binary search.

Rizzo further claimed[12] that 16K-1 can be fine tuned to use as few as 4 queries to recover an unknown character when $|\Omega| = 64$, which may require a different method of selecting Q than what we describe, e.g. sampling each character of Q according to a skewed distribution on cands.

Divide and conquer

Despite not having been covered in [19, 48], the divide-and-conquer technique for compression side-channel attacks is often associated with CRIME and BREACH and is sometimes confused with the 16K-1 technique. This

[12] https://twitter.com/julianor/status/245943430570704896

technique was likely introduced by Thomas Pornin and later rediscovered in [26].

The divide-and-conquer technique is similar to 16K-1, except for the choice of Q and the way of updating the candidate set cands. Suppose the candidate set cands for the first unknown character in ck is currently $\{g_1, \ldots, g_m\}$ for some $m > 1$; the attacker makes a query Q of the form

$$P \| g_1 \| \mathsf{filler}_1 \| P \| g_2 \| \mathsf{filler}_2 \ldots \| P \| g_{\lfloor m/2 \rfloor}$$

where $\mathsf{filler}_1, \ldots, \mathsf{filler}_{\lfloor m/2 \rfloor - 1}$ are optional fillers to separate different guesses. If the first unknown character in ck is in $\{g_1, \ldots, g_{\lfloor m/2 \rfloor}\}$, then Q contains a repeated string $P \| g_i$ for some $i \in [\lfloor m/2 \rfloor]$ in addition to the prefixes P, so its corresponding plaintext is likely compressed to a slightly smaller length. Therefore, the attacker sets cands to $\{g_1, \ldots, g_{\lfloor m/2 \rfloor}\}$ if the ciphertext is shorter, and to $\{g_{\lfloor m/2 \rfloor + 1}, \ldots, g_m\}$ if the ciphertext is longer. In practice, the attacker may need to use a variant of the Two Tries method to make the signal visible. The expected number of queries is in theory around $\ell \lceil \log |\Sigma| \rceil$.

The divide-and-conquer technique can potentially be used to improve many compression side-channel attacks. However, this technique likely suffers from greater instability, which partly offsets the reduced number of queries.

Chapter 4

Theoretical bounds

4.1 Overview

The main goal of this chapter is to derive upper bounds on the advantage of an attacker in the cookie recovery (CR) game (Section 3.2.5).

In Section 4.2, we introduce two new security notions, the length-only cookie recovery (LOCR) security and the length-only cookie indistinguishability (LOCI) security. These two notions characterise the advantage for an unbounded adversary to recover a secret cookie or distinguish between two cookies from the compressed lengths of the cookie and the adaptively chosen inputs. Assuming the encryption scheme is IND-CPA secure, we use standard techniques to derive a reduction from LOCR to CR in Theorem 4.1.

In Section 4.3, we bound the LOCR security of a compressor using its sensitivity, which is the maximum length difference of the compressed data with regard to the substitution of a single character. Intuitively, if the sensitivity of a compressor is very low, then each oracle answer in the LOCR game only contains a small amount of information. We formalise this intuition in Theorem 4.10. However, most compressors studied in [1] have a high sensitivity, and we further prove in Appendix A that the existence of a highly compressible string implies a non-constant sensitivity.

In Section 4.4, we bound the LOCR security of a compressor with its LOCI security. Since our definition of the LOCI game hard-codes a pair of cookies, we are able to fine-tune the bound we get in Lemma 4.19 by adjusting the probability of each pair of cookies, similar to the coupling technique. As a result, Lemma 4.19 unifies two different theorems: Theorem 4.21 bounds the LOCR security with a dominating set, where every cookie is indistinguishable to a cookie in the dominating set, while Theorem 4.23 and Corollary 4.25 bound the LOCR security by considering the ratio between the probability of a cookie and the collective probability of cookies indistinguishable to that cookie.

In Section 4.5, we apply Theorem 4.21 to analyse the LOCR security of two simple compressors, the fixed-dictionary compressor defined in [2] and Huffman coding. In particular, while the CR security of the fixed-dictionary compressor was already analysed in [2], we point out that their security analysis contains several flaws and provide a new proof.

In Section 4.6, we perform a case study of the effect of padding on compression side-channel attacks. Degabriele [9] studied random padding schemes for length-hiding encryption, and demonstrated the superior performance of Gaussian padding over uniform padding by evaluating its effect on a variant of the CRIME attack. We point out a gap between the difference hiding game and the multisample distinguisher theorem (Theorem 3.10) in [9], and bridge the gap with similar techniques as [7,8]. Then, we bound the LOCR security with the padding scheme by applying a corollary of Theorem 4.23.

As the best known compression side-channel attack requires $\Theta(\ell \log |\Omega|)$ attempts to recover a secret cookie in Ω^{ℓ}, and compression side-channel attacks are generally sensitive to noise, some of the results in this chapter may appear counter-intuitive. They are nevertheless justified by our refined techniques for compression side-channel attacks in Chapters 5 and 6.

4.2 New security notions

In this section, we present two new security notions for a compressor, the length-only cookie recovery (LOCR) security and the length-only cookie indistinguishability (LOCI) security.

4.2.1 Length-only cookie recovery (LOCR)

Intuitively, in the CR security game, if the scheme is a compress-then-encrypt construction (Definition 3.9) with an underlying IND-CPA secure encryption scheme, then an adversary cannot learn much more than compressed lengths. The bound in [2, Theorem 2] suggested that Alawatugoda et al. have used this intuition, but we were unable to find any corresponding proof or argument in their paper. Similar results on "removing" encryption were also present in [17,55].

For completeness, we formalise this intuition and give a proof using standard techniques.

We define the length-only cookie recovery (LOCR) game in Fig. 4.1. The advantage of $\mathcal{A}$ is

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L}(\mathcal{A}) := \Pr[\mathsf{LOCR}(\mathcal{A}, C, \mathcal{D}_{\mathcal{CK}}, L) \Rightarrow \mathsf{true}].$$

```
Game LOCR($\mathcal{A}, C, \mathcal{D}_{\mathcal{CK}}, L$)
1 :  $ck \leftarrow\!\!\$\ \mathcal{D}_{\mathcal{CK}}$
2 :  $ck' \leftarrow\!\!\$\ \mathcal{A}^{O}()$
3 :  return $ck = ck'$
```

```
Oracle $O(m', m'')$
1 :  $m \leftarrow m' \| ck \| m''$
2 :  if $|m| > L$ then
3 :      return $\perp$
4 :  $z \leftarrow\!\!\$\ C(m)$
5 :  return $|z|$
```

Figure 4.1: The LOCR security game [2] for compressor C on Σ, cookie distribution $\mathcal{D}_{\mathcal{CK}}$ on $\mathcal{CK} \subseteq \Sigma^k$, and maximum length L, where $k, L \in \mathbb{N}_+$ and $k \leq L$. The adversary $\mathcal{A}$ can be computationally unbounded.

The LOCR security of compressor C with regard to cookie space $\mathcal{D}_{\mathcal{CK}}$, maximum length L and number of queries $q \in \mathbb{N}$ is

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{CK},L,q} := \sup\{\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{CK},L}(\mathcal{A}) : \forall \mathcal{A} \text{ s.t. } \mathcal{A} \text{ makes at most } q \text{ queries to } O\}.$$

Theorem 4.1 *Let $\Gamma = (C, D)$ be a compresion scheme on input alphabet Σ, and let $\Pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key encryption scheme over message space $\mathcal{W} \subseteq \Omega^+$. For simplicity, we assume that $\exists \omega \in \Omega$, such that $\forall s \in \mathcal{W}$, $\omega^{|s|} \in \mathcal{W}$.*

Let $\Pi' = (\mathsf{KGen}', \mathsf{Enc}', \mathsf{Dec}')$ be a compress-then-encrypt construction of Γ and Π over message space $\mathcal{M} \subseteq \Sigma^+$. Let $k, L \in \mathbb{N}_+$ such that $k \leq L$ and $\bigcup_{i=k}^{L} \Sigma^i \subseteq \mathcal{M}$. Let $\mathcal{D}_{\mathcal{CK}}$ be a cookie distribution on $\mathcal{CK} \subseteq \Sigma^k$.

For every CR *adversary $\mathcal{A}$ that makes at most q queries to oracles E_1 and E_2, we can construct an* IND-CPA *adversary $\mathcal{B}$ against Π that makes at most q queries to the oracle* LoR, *such that*

$$\mathsf{Adv}^{\mathsf{CR}}_{\Pi',\mathcal{D}_{\mathcal{CK}},L}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{CK},L,q}.$$

Remark 4.2 *Our construction of $\mathcal{B}$ runs the compressor C at most q times and samples once from $\mathcal{D}_{\mathcal{CK}}$. For the derived bound to be meaningful in the real world, (i) the compressor C should be efficient, and (ii) the cookie distribution $\mathcal{D}_{\mathcal{CK}}$ should be efficiently sampleable.*

Proof Without loss of generality, assume that $\mathcal{A}$ makes at most q oracle queries, and that $\mathcal{A}$ only makes valid queries, i.e. for every query $E_1(m', m'')$ that $\mathcal{A}$ makes, $m', m'' \in \Sigma^*$, and $|m'| + |m''| \leq L - k$, and for every query $E_2(m)$ that $\mathcal{A}$ makes, $m \in \mathcal{M}$.

Let game $G_0(\cdot) = \mathsf{CR}(\cdot, \Pi', \mathcal{D}_{\mathcal{CK}}, L)$, and game G_1 be as in Fig. 4.2.

We construct an IND-CPA adversary $\mathcal{B}$ against Π from $\mathcal{A}$ as in Fig. 4.3. Since $\mathcal{A}$ makes at most q queries, $\mathcal{B}$ also makes at most q queries to LoR. The adversary $\mathcal{B}$ does not make invalid queries, because $z \in \mathcal{W}$ by the definition of

```
Game $G_1(\mathcal{A})$
1 : $k \leftarrow\!\!\$\ \mathsf{KGen}'()$
2 : $ck \leftarrow\!\!\$\ \mathcal{D}_{\mathcal{CK}}$
3 : $ck' \leftarrow\!\!\$\ \mathcal{A}^{E_1,E_2}()$
4 : return $ck = ck'$

Oracle $E_1(m', m'')$
1 : $m \leftarrow m' \| ck \| m''$
2 : if $|m| > L$ then
3 :     return $\bot$
4 : $z \leftarrow\!\!\$\ C(m)$
5 : $c \leftarrow\!\!\$\ \mathsf{Enc}(k, \omega^{|z|})$
6 : return $c$

Oracle $E_2(m)$
1 : $c \leftarrow\!\!\$\ \mathsf{Enc}'(k, m)$
2 : return $c$
```

Figure 4.2: The game G_1. We modify oracle E_1 to return the encryption of $\omega^{|z|}$.

```
Adversary $\mathcal{B}^{\mathsf{LoR}}()$
1 : $ck \leftarrow\!\!\$\ \mathcal{D}_{\mathcal{CK}}$
2 : $ck' \leftarrow\!\!\$\ \mathcal{A}^{E_1,E_2}()$
3 : if $ck = ck'$ then
4 :     return 1
5 : else
6 :     return 0

Oracle $E_1(m', m'')$
1 : $m \leftarrow m' \| ck \| m''$
2 : if $|m| > L$ then
3 :     return $\bot$
4 : $z \leftarrow\!\!\$\ C(m)$
5 : $c \leftarrow\!\!\$\ \mathsf{LoR}(z, \omega^{|z|})$
6 : return $c$

Oracle $E_2(m)$
1 : $z \leftarrow\!\!\$\ C(m)$
2 : $c \leftarrow\!\!\$\ \mathsf{LoR}(z, z)$
3 : return $c$
```

Figure 4.3: The adversary $\mathcal{B}$.

compress-then-encrypt, $\omega^{|z|} \in \mathcal{W}$ by our assumption, and $|z| = |\omega^{|z|}|$. The answer to the oracle query $\mathsf{LoR}(z, z)$ has the same distribution as $\mathsf{Enc}(k, z)$.

If $b = 0$ in the IND-CPA game, then $\mathsf{LoR}(z, \omega^{|z|})$ has the same distribution as $\mathsf{Enc}(k, z)$; therefore, $\mathcal{B}$ simulates E_1 and E_2 perfectly for $\mathcal{A}$ in game G_0, and thus $\Pr[\mathcal{B}() \Rightarrow 1 \mid b = 0] = \Pr[G_0(\mathcal{A}) \Rightarrow \mathsf{true}]$. Otherwise, if $b = 1$ in the IND-CPA game, then $\mathsf{LoR}(z, \omega^{|z|})$ has the same distribution as $\mathsf{Enc}(k, \omega^{|z|})$, so $\mathcal{B}$ simulates E_1 and E_2 perfectly for $\mathcal{A}$ in game G_1; therefore, we have $\Pr[\mathcal{B}() \Rightarrow 1 \mid b = 1] = \Pr[G_1(\mathcal{A}) \Rightarrow \mathsf{true}]$. The advantage of $\mathcal{B}$ satisfies

$$\begin{aligned}\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi}(\mathcal{B}) &= |\Pr[\mathcal{B}() \Rightarrow 1 \mid b = 0] - \Pr[\mathcal{B}() \Rightarrow 1 \mid b = 1]| \\ &= |\Pr[G_0(\mathcal{A}) \Rightarrow \mathsf{true}] - \Pr[G_1(\mathcal{A}) \Rightarrow \mathsf{true}]|,\end{aligned}$$

so

$$\Pr[G_0(\mathcal{A}) \Rightarrow \mathsf{true}] \leq \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi}(\mathcal{B}) + \Pr[G_1(\mathcal{A}) \Rightarrow \mathsf{true}]. \tag{4.1}$$

Define $G_2(\cdot) = \mathsf{LOCR}(\cdot, C, \mathcal{D}_{\mathcal{CK}}, L)$. We build a G_2 adversary $\mathcal{C}$ on a G_1 adversary $\mathcal{A}$, as in Fig. 4.4.

By our assumptions on $\mathcal{A}$, the G_2 adversary $\mathcal{C}$ makes at most q queries to the oracle O, and simulates game G_1 perfectly for $\mathcal{A}$. Therefore,

$$\Pr[G_1(\mathcal{A}) \Rightarrow \mathsf{true}] = \Pr[G_2(\mathcal{C}) \Rightarrow \mathsf{true}] \leq \mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L,q}. \tag{4.2}$$

```
Adversary $\mathcal{C}^{O}()$
1 : $k \leftarrow\$ \mathsf{KGen}()$
2 : $ck' \leftarrow\$ \mathcal{A}^{E_1,E_2}()$
3 : return $ck'$
```

```
Oracle $E_1(m', m'')$
1 : $l \leftarrow O(m', m'')$
2 : $c \leftarrow\$ \mathsf{Enc}(k, \omega^{l})$
3 : return $c$
```

```
Oracle $E_2(m)$
1 : $z \leftarrow\$ C(m)$
2 : $z \leftarrow\$ \mathsf{Enc}(k, z)$
3 : return $c$
```

Figure 4.4: The adversary $\mathcal{C}$.

Combining Eq. (4.1) and Eq. (4.2), we get

$$\mathsf{Adv}^{\mathsf{CR}}_{\Pi',\mathcal{D}_{\mathcal{CK}},L}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L,q}. \qquad \square$$

4.2.2 Length-only cookie indistinguishability (LOCI)

Alawatugoda et al. also defined two indistinguishability-style security notions, the random cookie indistinguishability (RCI) security and the chosen cookie indistinguishability (CCI) security [2]. The CCI security requires that it should be infeasible for an adversary to distinguish between two cookies of their choice is in the plaintext, and the RCI security requires that it should be infeasible for an adversary to distinguish between two randomly chosen cookies known to the adversary. Alawatugoda et al. showed a sequence of reductions IND-CPA $\Rightarrow$ CCI $\Rightarrow$ RCI $\Rightarrow$ CR and gave separating examples.

However, the CCI security can hardly be achieved by any non-trivial deterministic compressor without being programmed to identify the cookie, and the benefit of random RCI security is not clear except for bounding the CR security. Therefore, we will not use these two security games in this thesis. Instead, we will base our results on a more fine-grained indistinguishability game.

```
Game LOCI$(\mathcal{A}, C, ck_0, ck_1, L)$
1 : $b \leftarrow\$ \{0,1\}$
2 : $b' \leftarrow\$ \mathcal{A}^{O}(ck_0, ck_1)$
3 : return $b = b'$
```

```
Oracle $O(m', m'')$
1 : $m \leftarrow m' \| ck_b \| m''$
2 : if $|m| > L$ then
3 :     return $\perp$
4 : $z \leftarrow\$ C(m)$
5 : return $|z|$
```

Figure 4.5: The LOCI security game for compressor C on Σ, cookies $ck_0, ck_1 \in \Sigma^{k}$, and maximum length L, where $k, L \in \mathbb{N}_{+}$, and $k \leq L$. The adversary $\mathcal{A}$ can be computationally unbounded.

We define the *length-only cookie indistinguishability* (LOCI) game in Fig. 4.5.

The advantage of $\mathcal{A}$ is

$$\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L}(\mathcal{A}) = |2\Pr[\mathsf{LOCI}(\mathcal{A}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] - 1|.$$

We define the LOCI security of compressor C with regard to cookies ck_0, ck_1, maximum length L and number of queries $q \in \mathbb{N}$ as

$$\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} = \sup\{\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L}(\mathcal{A}) : \forall\mathcal{A} \text{ s.t. } \mathcal{A} \text{ makes at most } q \text{ queries to } O\}.$$

We point out that the LOCI security degenerates to a binary value when the compressor C is deterministic.

Theorem 4.3 *Let C be a compressor, $k, L \leq \mathbb{N}_+$ such that $k \leq L$, and let $ck_0, ck_1 \in \Sigma^k$. If C is deterministic, then for all $q \in \mathbb{N}$, $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \in \{0, 1\}$.*

Proof See Appendix B.1. □

4.3 Sensitivity

To simplify discussion, we first introduce an artificial concept called the coin space. Without loss of generality, we assume that a compressor C uses the same amount of coin tosses when handling all inputs of the same length.

Definition 4.4 (Coin space) *Let C be a compressor, and $\ell \in \mathbb{N}_+$. The coin space $\mathcal{R}_{C,\ell}$ is the set of random coins that C use for inputs of length ℓ.*

Remark 4.5 *Note that the coin space $\mathcal{R}_{C,\ell}$ is finite by definition.*

We adapt the definition of additive sensitivity with regard to substitution by Akagi et al. [1, Section 2.2], and extend their definition to also apply to randomised compressors as well as more than one possible substitutions.

Definition 4.6 ((k-)sensitivity) *Let C be a compressor on Σ, and $\ell, k \in \mathbb{N}_+$. The k-sensitivity of C at length ℓ is*

$$\Delta_k(C, \ell) = \max_{r \in \mathcal{R}_{C,\ell}} \max_{\substack{s,s' \in \Sigma^\ell \\ \mathsf{HD}(s,s') \leq k}} \left(\left|C(s'; r)\right| - |C(s; r)|\right). \tag{4.3}$$

If C is deterministic, then Eq. (4.3) *can be simplified to*

$$\Delta_k(C, \ell) = \max_{\substack{s,s' \in \Sigma^\ell \\ \mathsf{HD}(s,s') \leq k}} \left(\left|C(s')\right| - |C(s)|\right).$$

We refer to 1-sensitivity directly as sensitivity and let $\Delta(C, \ell) = \Delta_1(C, \ell)$.

We now prove some basic results on k-sensitivity.

Lemma 4.7 *Let C be a compressor on Σ, $\ell \in \mathbb{N}_+$, $r \in \mathcal{R}_{C,\ell}$. For all $a, b \in \Sigma^\ell$,*

$$||C(b;r)| - |C(a;r)|| \leq \mathsf{HD}(a,b) \cdot \Delta(C,\ell).$$

Proof Let $k = \mathsf{HD}(a,b)$, and let $i_1 < i_2 < \cdots < i_k$ be the positions where a and b differ. Let $s_0 = b$, and for $j \in [k]$, let $s_j = a_1 \dots a_{i_j} b_{i_j+1} \dots b_l \in \Sigma^\ell$. By definition, $s_k = a_1 \dots a_{i_k} b_{i_k+1} \dots b_l = a_1 \dots a_{i_k} a_{i_k+1} \dots a_l = a$, and for $\forall j \in [k]$, $\mathsf{HD}(s_{j-1}, s_j) = 1$. Therefore,

$$\begin{aligned} ||C(b;r)| - |C(a;r)|| &= ||C(s_k;r)| - |C(s_0;r)|| \\ &= \left| \sum_{i=1}^{k} (|C(s_i;r)| - |C(s_{i-1};r)|) \right| \\ &\leq \sum_{i=1}^{k} |(|C(s_i;r)| - |C(s_{i-1};r)|)| \leq k \cdot \Delta(C,\ell). \qquad \square \end{aligned}$$

Corollary 4.8 *Let C be a compressor, $\ell \in \mathbb{N}_+$, $r \in \mathcal{R}_{C,\ell}$. We have*

$$\Delta(C,\ell) \geq \frac{1}{\ell} \left(\max_{s \in \Sigma^\ell} |C(s;r)| - \min_{s \in \Sigma^\ell} |C(s;r)| \right).$$

Lemma 4.9 *Let C be a compressor, $\ell \in \mathbb{N}_+$. The following claims hold:*

a) For all $k_1, k_2 \in \mathbb{N}_+$ where $k_1 \leq k_2$, we have $\Delta_{k_1}(C,\ell) \leq \Delta_{k_2}(C,\ell)$;

b) For all $k \in \mathbb{N}_+$, we have $\Delta_k(C,\ell) \leq k \cdot \Delta(C,\ell)$;

Proof Claim a) holds by the definition of k-sensitivity.

For claim b), note that

$$\begin{aligned} \Delta_k(C,\ell) &= \max_{r \in \mathcal{R}_{C,\ell}} \max_{\substack{s,s' \in \Sigma^\ell \\ \mathsf{HD}(s,s') \leq k}} (|C(s';r)| - |C(s;r)|) \\ &\leq \max_{r \in \mathcal{R}_{C,\ell}} \max_{\substack{s,s' \in \Sigma^\ell \\ \mathsf{HD}(s,s') \leq k}} \mathsf{HD}(s',s) \cdot \Delta(C,\ell) \qquad \text{(Lemma 4.7)} \\ &\leq k \cdot \Delta(C,\ell). \qquad \square \end{aligned}$$

4.3.1 Generic bound

Theorem 4.10 *Let C be a compressor on Σ, let $k, L \in \mathbb{N}_+$ such that $k \leq L$, let $\mathcal{D}_{\mathcal{CK}}$ be a cookie distribution on $\mathcal{CK} \subseteq \Sigma^k$, and let*

$$\delta_{k,L} = \max_{\ell \in \{k,\dots,L\}} \Delta_k(C,\ell).$$

For every $q \in \mathbb{N}$, we have

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L,q} \leq \max_{\substack{S \subseteq \mathcal{CK} \\ |S| \leq (1+\delta_{k,L})^q}} \Pr_{\mathcal{D}_{\mathcal{CK}}}[S].$$

Remark 4.11 *One may also apply Fano's inequality to obtain a likely worse bound.*

The intuition is that if we assume that the compressor is deterministic, then each oracle query only has $\delta_{k,L} + 1$ possible outcomes. An adversary maps $(\delta_{k,L} + 1)^q$ inputs to $|\mathcal{CK}|$ possible outputs in the end, and the proof is done by a direct argument. We extend the intuition to all compressors via a simple derandomisation, where we fix the random coins for answering oracle queries and give them to the adversary beforehand.

Proof Let $\mathcal{A}$ be an LOCR adversary that makes at most q queries.

Without loss of generality, we assume that $\mathcal{A}$ makes exactly q queries to O in the LOCR game, and that $\mathcal{A}$ does not make invalid queries, i.e. for every query $O(m', m'')$ that $\mathcal{A}$ makes, $m', m'' \in \Sigma^*$, and $|m'| + |m''| \leq L - k$.

```
Game $G(\mathcal{A}, q)$
1: $i_q \leftarrow 0$
2: for $j = k, \ldots, L$ do
3:     $r_{1,j} \ldots r_{q,j} \leftarrow\!\!\$\ \mathcal{R}_{C,j}$
4: $R \leftarrow \{r_{i,j}\}_{i \in [q], j \in \{k,\ldots,L\}}$
5: $ck \leftarrow\!\!\$\ \mathcal{D}_{\mathcal{CK}}$
6: $ck' \leftarrow\!\!\$\ \mathcal{A}^O(R)$
7: return $ck = ck'$
```

```
Oracle $O(m', m'')$
1: $i_q \leftarrow i_q + 1$
2: $m \leftarrow m' \| ck \| m''$
3: $r \leftarrow r_{i_q, |m|}$
4: $c \leftarrow C(m; r)$
5: $\ell_0 \leftarrow \min_{ck' \in \mathcal{CK}} |C(m' \| ck' \| m''; r)|$
6: return $|c| - \ell_0$
```

Figure 4.6: Game G. Note that the oracle O does not use any fresh random coins.

Consider the game G defined in Fig. 4.6. Game G selects all randomness it may use when answering oracle queries, and gives it to the adversary $\mathcal{A}$. When $\mathcal{A}$ queries O on input m', m'', game G derandomises the compressor C with the corresponding random coins, and computes a "differential" compressed length, $|c| - \ell_0$. By definition, $|c| - \ell_0 \leq \max_{\ell \in \{k,\ldots,L\}} \Delta_k(C, \ell) = \delta_{k,L}$.

```
Game $\mathcal{B}^O(R)$
1: $\{r_{i,j}\}_{i \in [q], j \in \{k,\ldots,L\}} \leftarrow R$
2: $i_q \leftarrow 0$
3: $ck' \leftarrow\!\!\$\ \mathcal{A}^{O'}()$
4: return $ck'$
```

```
Oracle $O'(m', m'')$
1: $i_q \leftarrow i_q + 1$
2: $d \leftarrow O(m', m'')$
3: $j \leftarrow |m'| + |m''| + k$
4: $\ell_0 \leftarrow \min_{ck' \in \mathcal{CK}} \left|C(m' \| ck \| m''; r_{i_q, j})\right|$
5: return $\ell_0 + d$
```

Figure 4.7: Adversary $\mathcal{B}$.

We construct an adversary $\mathcal{B}$ playing in game G from $\mathcal{A}$, as in Fig. 4.7. For each oracle query made by the LOCR adversary $\mathcal{A}$, the adversary $\mathcal{B}$ queries its own oracle and uses the randomness it received at the beginning of the game to recover the compressed length. Therefore, $\mathcal{B}$ simulates the LOCR game perfectly for $\mathcal{A}$, and

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L}(\mathcal{A}) = \Pr[G(\mathcal{B},q) \Rightarrow \mathsf{true}]. \tag{4.4}$$

Let $\mathcal{B}'$ be obtained by fixing the random coins used by $\mathcal{B}$ to maximise the probability that game G returns true. By definition, $\mathcal{B}'$ is deterministic, and

$$\Pr[G(\mathcal{B},q) \Rightarrow \mathsf{true}] \leq \Pr\left[G(\mathcal{B}',q) \Rightarrow \mathsf{true}\right]. \tag{4.5}$$

For $i \in [q]$, let d_i denote the answer to the i-th oracle query to O'. We observe that for adversary $\mathcal{B}'$ in game G, the final output ck' only depends on R and $d_1, \ldots, d_q$, which we can write as

$$ck' = f_{B'}(R, d_1, \ldots, d_q).$$

Let $E_{(d_1,\ldots,d_q)}$ denote the event that the answers to the oracle queries made by $\mathcal{B}$ are $d_1, \ldots, d_q$, respectively. Let R be arbitrarily fixed. Let

$$S'(R) = \left\{f_{B'}(R, d_1, \ldots, d_q) : d_1, \ldots, d_q \in \{0, \ldots, \delta_{k,L}\}\right\}.$$

By definition, $|S'(R)| \leq (1+\delta_{k,L})^q$. For all $ck \in \mathcal{CK}$, we have

$$\begin{aligned}
&\Pr\left[G(\mathcal{B}',q) \Rightarrow \mathsf{true} \,\middle|\, R, ck\right] \\
&= \sum_{d_1,\ldots,d_q \in \{0,\ldots,\delta_{k,L}\}} \Pr\left[E_{(d_1,\ldots,d_q)} \,\middle|\, R, ck\right] \Pr\left[\mathcal{B}(R) \Rightarrow ck \,\middle|\, E_{(d_1,\ldots,d_q)}\right] \\
&\leq \max_{d_1,\ldots,d_q \in \{0,\ldots,\delta_{k,L}\}} \Pr\left[\mathcal{B}(R) \Rightarrow ck \,\middle|\, E_{(d_1,\ldots,d_q)}\right] \\
&= \mathbf{1}_{S'(R)}(ck).
\end{aligned}$$

Therefore,

$$\begin{aligned}
\Pr\left[G(\mathcal{B}',q) \Rightarrow \mathsf{true} \,\middle|\, R\right] &\leq \sum_{ck \in \mathcal{CK}} \mathbf{1}_{ck \in S'(R)} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck] \\
&= \Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S'(R)\right] \\
&\leq \sum_{\substack{S \subseteq \mathcal{CK} \\ |S| \leq (1+\delta_{k,L})^q}} \Pr_{\mathcal{D}_{\mathcal{CK}}}[S].
\end{aligned} \tag{4.6}$$

Applying the law of total probability on Eq. (4.6), we get

$$\Pr\left[G(\mathcal{B}',q) \Rightarrow \mathsf{true}\right] \leq \sum_{\substack{S \subseteq \mathcal{CK} \\ |S| \leq (1+\delta_{k,L})^q}} \Pr_{\mathcal{D}_{\mathcal{CK}}}[S]. \tag{4.7}$$

Finally, combining Eq. (4.4), Eq. (4.5) and Eq. (4.7) completes the proof. □

Corollary 4.12 *For every $q \in \mathbb{N}$,*

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L,q} \leq (1+\delta_{k,L})^q \max_{ck \in \mathcal{CK}} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck].$$

Equivalently, for every LOCR *adversary $\mathcal{A}$, let q be the maximum number of queries that $\mathcal{A}$ makes. We have*

$$q \geq \frac{\log \mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L}(\mathcal{A}) - \log \max_{ck \in \mathcal{CK}} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck]}{\log(1+\delta_{k,L})}.$$

Remark 4.13 *This corollary can also be proved directly by modifying our proof of Theorem 4.10, where we simulate game G for the adversary $\mathcal{B}$ by answering its oracle queries with integers sampled uniformly at random from $\{0,\ldots,\delta_{k,L}\}$.*

Corollary 4.14 *Let $\mathcal{U}_{\mathcal{CK}}$ be a uniform distribution on $\mathcal{CK} \subseteq \Sigma^k$. For every $q \in \mathbb{N}$,*

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{U}_{\mathcal{CK}},L,q} \leq \frac{(1+\delta_{k,L})^q}{|\mathcal{CK}|},$$

Equivalently, for every LOCR *adversary $\mathcal{A}$, let q be the maximum number of queries that $\mathcal{A}$ makes. We have*

$$q \geq \frac{\log \mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{U}_{\mathcal{CK}},L}(\mathcal{A}) + \log|\mathcal{CK}|}{\log(1+\delta_{k,L})}.$$

Remark 4.15 *If $\delta_{k,L}$ is a small constant, then the adversary has to make $O(\log|\mathcal{CK}|)$ queries to succeed with a high probability, which can very roughly be imitated with a binary search.*

4.3.2 Negative results

Theorem 4.10 and its corollaries may give the impression that compressors with low constant sensitivities should be preferred in order to mitigate compression side-channel attacks. However, most compressors considered by Akagi et al. [1] have a sensitivity of $\Omega(\sqrt{\ell})$, making our results much less useful, especially if the maximum length ℓ is large.

We further argue in Appendix A that a constant sensitivity is out of reach for most compressors. For example, one of our results shows that for compressor C, $\alpha \in (0,1)$ and $\ell \in \mathbb{N}$, if there exists a string of length ℓ that can be compressed by C to a string of length $O(\ell^\alpha)$, then $\Delta(C,\ell) = \Omega(\log \ell)$. We refer interested readers to Appendix A for more details.

On one hand, the negative results suggest the need for a more refined security metric. On the other hand, they hint that it is possible to design attacks that require fewer queries than the divide-and-conquer attack by making specifically crafted queries, which we will show in subsequent chapters.

4.4 From LOCI to LOCR

4.4.1 Indistinguishable sets

Definition 4.16 ((q, ε)-indistinguishable) *Let C be a compressor on Σ, $k, L \in \mathbb{N}_+$ such that $k \leq L$, $\mathcal{CK} \subseteq \Sigma^k$. For $q \in \mathbb{N}$ and $0 \leq \varepsilon \leq 1$, two cookies $ck_0, ck_1 \in \mathcal{CK}$ are (q, ε)-indistinguishable if $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \leq \varepsilon$.*

We call ck_0 and ck_1 ε-indistinguishable if $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \leq \varepsilon$ for all $q \in \mathbb{N}$.

Definition 4.17 (Indistinguishable set) *Let C be a compressor on Σ, $k, L \in \mathbb{N}_+$ such that $k \leq L$, $\mathcal{CK} \subseteq \Sigma^k$. For $ck \in \mathcal{CK}$, the (q, ε)-indistinguishable set of ck is*

$$S_{C,\mathcal{CK},L,q,\varepsilon}(ck) = \{ck' \in \mathcal{CK} : \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck,ck',L,q} \leq \varepsilon\}.$$

We may omit C, $\mathcal{CK}$ and L from the subscript if they are clear from the context.

Note that both definitions do not rely on the cookie distribution $\mathcal{D}_{\mathcal{CK}}$.

The relation $ck_0 \in S_{q,\varepsilon}(ck_1)$ is reflexive and symmetric, but not transitive. However, it is clear that if $ck_0 \in S_{q_1,\varepsilon_1}(ck_1)$ and $ck_1 \in S_{q_2,\varepsilon_2}(ck_2)$, then $ck_0 \in S_{\max\{q_1,q_2\},\varepsilon_1+\varepsilon_2}(ck_2)$.

Lemma 4.18 *Let C be a compressor on Σ, $k, L \in \mathbb{N}_+$ such that $k \leq L$, $\mathcal{CK} \subseteq \Sigma^k$, $q \in \mathbb{N}$, $0 \leq \varepsilon \leq 1$. For all $ck_0, ck_1 \in \mathcal{CK}$, if $ck_0 \in S_{q,\varepsilon}(ck_1)$, then $ck_1 \in S_{q,\varepsilon}(ck_0)$.*

Proof Suppose, for contradiction, that there exists $ck_0, ck_1 \in \mathcal{CK}$, such that $ck_0 \in S_{q,\varepsilon}(ck_1)$, and $ck_1 \notin S_{q,\varepsilon}(ck_0)$. By definition, $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \leq \varepsilon$, and $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_1,ck_0,L,q} > \varepsilon$. Therefore, there exists a LOCI adversary $\mathcal{A}$ against C, ck_1, ck_0, L that makes at most q queries, such that

$$\varepsilon < \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_1,ck_0,L}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_1,ck_0,L,q}.$$

Let LOCI adversary $\mathcal{A}'$ against C, ck_0, ck_1, L be obtained by running $\mathcal{A}(ck_1, ck_0)$ to obtain a guess b', and then returning $1 - b'$. By definition, $\mathcal{A}'$ also makes at most q queries. We have

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} &\geq \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L}(\mathcal{A}') \\
&= \left|2\Pr\left[\mathsf{LOCI}(\mathcal{A}', C, ck_0, ck_1, L) \Rightarrow \mathsf{true}\right] - 1\right| \\
&= |2\Pr[\mathsf{LOCI}(\mathcal{A}, C, ck_1, ck_0, L) \Rightarrow \mathsf{true}] - 1| \\
&= \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_1,ck_q,L}(\mathcal{A}) > \varepsilon.
\end{aligned}$$

We arrived at a contradiction since $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \leq \varepsilon$. □

4.4.2 Bounding LOCR

Lemma 4.19 *Let C be a compressor on Σ, let $k, L \in \mathbb{N}_+$ such that $k \leq L$, and let $\mathcal{D}_{\mathcal{CK}}$ be a cookie distribution on $\mathcal{CK} \subseteq \Sigma^k$. Let $q \in \mathbb{N}$ and $0 \leq \varepsilon \leq 1$.*

Let $\mathcal{A}$ be a LOCR *adversary against $C, \mathcal{D}_{\mathcal{CK}}$ and L that makes at most q queries. For adversary $\mathcal{A}$, let $\{\mathcal{P}_{ck}\}_{ck \in \mathcal{CK}}$ be a set of distributions on $\mathcal{CK}$, such that for every $ck \in \mathcal{CK}$, we have $\mathrm{Supp}(\mathcal{P}_{ck}) \subseteq S_{q,\varepsilon}(ck)$. Then*

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{CK},L}(\mathcal{A}) \leq \varepsilon' + \sum_{ck_0 \in \mathcal{CK}} \max_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1],$$

where

$$\varepsilon' = \sum_{ck_0 \in \mathcal{CK}} \sum_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \leq \varepsilon.$$

Remark 4.20 *The distributions $\{\mathcal{P}_{ck}\}_{ck \in \mathcal{CK}}$, as we will show, are artificial concepts that have no concrete meaning in the real world and are only used to simplify our proof for subsequent theorems. They can be selected arbitrarily and do not need to be efficiently sampleable; the selection of distributions may also depend on the adversary $\mathcal{A}$, in addition to other parameters.*

The intuition is that an adversary should not be able to do much better than randomly guessing an element in $S_{q,\varepsilon}(ck)$ to recover ck. However, there are some caveats that require careful treatment.

Proof Without loss of generality, we assume that $\mathcal{A}$ makes exactly q queries. We construct a LOCI adversary $\mathcal{B}$ from $\mathcal{A}$ as in Fig. 4.8. Note that we use the same construction of $\mathcal{B}$ for all $ck_0, ck_1 \in \mathcal{CK}$.

```
Adversary $\mathcal{B}^{O}(ck_0, ck_1)$
1 : $ck' \leftarrow\$ \mathcal{A}^{O}()$
2 : **if** $ck' = ck_1$ **then**
3 :     **return** 1
4 : **else**
5 :     **return** 0
```

Figure 4.8: Construction of the LOCI adversary $\mathcal{B}$.

The LOCI adversary $\mathcal{B}$ simulates the LOCR game for $\mathcal{A}$, also making exactly q queries to the oracle (Fig. 4.1). The adversary $\mathcal{B}$ returns 1 if $\mathcal{A}$ outputs ck_1, and returns 0 otherwise. The adversary $\mathcal{B}$ simulates perfectly the LOCR game for $\mathcal{A}$ conditioned on $ck = ck_b$ in the LOCR game.

By definition, for all $ck_0, ck_1 \in \mathcal{CK}$,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} &\geq \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L}(\mathcal{B})\\ &= |\Pr[\mathcal{B}(ck_0, ck_1) \Rightarrow 1 \mid b = 0] - \Pr[\mathcal{B}(ck_0, ck_1) \Rightarrow 1 \mid b = 1]|\\ &= \big|\Pr\big[ck' = ck_1 \;\big|\, b = 0\big] - \Pr\big[ck' = ck_1 \;\big|\, b = 1\big]\big|.\end{aligned}$$

Translating the above result to the LOCR game, we get

$$\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \geq |\Pr[\mathcal{A}() \Rightarrow ck_1 \mid ck = ck_0] - \Pr[\mathcal{A}() \Rightarrow ck_1 \mid ck = ck_1]|,$$

and it immediately follows that

$$\Pr[\mathcal{A}() \Rightarrow ck_1 \mid ck = ck_1] \leq \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} + \Pr[\mathcal{A}() \Rightarrow ck_1 \mid ck = ck_0]. \quad (4.8)$$

To simplify notation, let $p_{ck_1|ck_1}$ be a shorthand for $\Pr[\mathcal{A}() \Rightarrow ck_1 \mid ck = ck_1]$, and $p_{ck_1|ck_0}$ be a shorthand for $\Pr[\mathcal{A}() \Rightarrow ck_1 \mid ck = ck_0]$. Equation (4.8) is then equivalent to $p_{ck_1|ck_0} \leq \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} + p_{ck_1|ck_0}$.

For all $ck_0, ck_1 \in \mathcal{CK}$, if $ck_1 \notin S_{q,\varepsilon}(ck_0)$, then by Lemma 4.18, $ck_0 \notin S_{q,\varepsilon}(ck_1)$, and since $\mathrm{Supp}(\mathcal{P}_{ck_1}) \subseteq S_{q,\varepsilon}(ck_1)$, we have $\Pr_{\mathcal{P}_{ck_1}}[ck_0] = 0$. Therefore,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L}(\mathcal{A}) &= \sum_{ck_1 \in \mathcal{CK}} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] p_{ck_1|ck_1}\\ &= \sum_{ck_1 \in \mathcal{CK}} \sum_{ck_0 \in \mathcal{CK}} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] p_{ck_1|ck_1}\\ &= \sum_{ck_0 \in \mathcal{CK}} \sum_{ck_1 \in \mathcal{CK}} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] p_{ck_1|ck_1}\\ &= \sum_{ck_0 \in \mathcal{CK}} \sum_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] p_{ck_1|ck_1}. \quad (4.9)\end{aligned}$$

Applying Eq. (4.8) to Eq. (4.9), we get

$$\begin{aligned}\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L}(\mathcal{A}) &\leq \sum_{ck_0 \in \mathcal{CK}} \sum_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \left(\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} + p_{ck_1|ck_0}\right)\\ &= \varepsilon' + \sum_{ck_0 \in \mathcal{CK}} \sum_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] p_{ck_1|ck_0}\\ &\leq \varepsilon' + \sum_{ck_0 \in \mathcal{CK}} \max_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1].\end{aligned}$$

We complete the proof by showing that

$$\begin{aligned}\varepsilon' &= \sum_{ck_0 \in \mathcal{CK}} \sum_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q}\\ &\leq \sum_{ck_0 \in \mathcal{CK}} \sum_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \varepsilon\\ &\leq \sum_{ck_1 \in \mathcal{CK}} \sum_{ck_0 \in \mathcal{CK}} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \varepsilon\\ &= \sum_{ck_1 \in \mathcal{CK}} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \varepsilon = \varepsilon. \qquad \square\end{aligned}$$

Determining the optimal selection for $\{\mathcal{P}_{ck}\}_{ck\in\mathcal{CK}}$ is likely difficult without knowing the exact structure of indistinguishable sets. However, we can derive useful results by considering some special cases.

Theorem 4.21 (Dominating set bound) *Let C be a compressor on Σ, let $k, L \in \mathbb{N}_+$ such that $k \leq L$, and let $\mathcal{D}_{\mathcal{CK}}$ be a cookie distribution on $\mathcal{CK} \subseteq \Sigma^k$. Let $q \in \mathbb{N}$ and $0 \leq \varepsilon \leq 1$.*

For every set $T \subseteq \mathcal{CK}$ be such that

$$\bigcup_{ck\in T} S_{q,\varepsilon}(ck) = \mathcal{CK}.$$

We have

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L,q} \leq \varepsilon + \sum_{ck_0\in T} \max_{ck_1\in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1].$$

Remark 4.22 *Setting $T = \mathcal{CK}$ guarantees that $\bigcup_{ck\in T} S_{q,\varepsilon}(ck) = \mathcal{CK}$, but this case is not very interesting.*

A variant of this theorem may be used to prove Theorem 4.10.

Proof Let $t = |T|$. We write $T = \{ck_1, \ldots, ck_t\}$. For every $i \in [t]$, let

$$S'(ck_i) = S_{q,\varepsilon}(ck_i) \setminus \bigcup_{j=1}^{i-1} S_{q,\varepsilon}(ck_j).$$

It follows that $\bigcup_{ck\in T} S'(ck) = \mathcal{CK}$, and $\forall i,j \in [m]$ where $i \neq j$, we have $S'(ck_i) \cap S'(ck_j) = \emptyset$.

For all $ck_0, ck_1 \in \mathcal{CK}$, let

$$\Pr_{\mathcal{P}_{ck_1}}[ck_0] = \begin{cases} 1, & ck_0 \in T \wedge ck_1 \in S'(ck_0) \\ 0, & \text{otherwise} \end{cases}.$$

It is easy to see that the distributions $\{\mathcal{P}_{ck}\}_{ck\in\mathcal{CK}}$ are well-defined.

Applying Lemma 4.19, we get

$$\begin{aligned} \mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{CK},L,q} &\leq \varepsilon + \sum_{ck_0\in\mathcal{CK}} \max_{ck_1\in S_{q,\varepsilon}(ck_0)} \mathbf{1}_T(ck_0)\mathbf{1}_{S'(ck_0)}(ck_1) \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \\ &= \varepsilon + \sum_{ck_0\in\mathcal{CK}} \mathbf{1}_T(ck_0) \max_{ck_1\in S'(ck_0)} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \\ &= \varepsilon + \sum_{ck_0\in T} \max_{ck_1\in S'(ck_0)} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \\ &\leq \varepsilon + \sum_{ck_0\in T} \max_{ck_1\in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]. \qquad \square \end{aligned}$$

Theorem 4.23 (Ratio-based bound) *Let C be a compressor on Σ, let $k, L \in \mathbb{N}_+$ such that $k \leq L$, and let $\mathcal{D}_{\mathcal{CK}}$ be a cookie distribution on $\mathcal{CK} \subseteq \Sigma^k$. Let $q \in \mathbb{N}$ and $0 \leq \varepsilon \leq 1$. Then*

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{D}_{\mathcal{CK}},L,q} \leq \varepsilon' + \sum_{ck_0 \in \mathsf{Supp}(\mathcal{D}_{\mathcal{CK}})} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0] \max_{ck_1 \in S_{q,\varepsilon}(ck_0)} \frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]},$$

where

$$\varepsilon' = \sum_{ck_0 \in \mathsf{Supp}(\mathcal{D}_{\mathcal{CK}})} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0] \sum_{ck_1 \in S_{q,\varepsilon}(ck_0)} \frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]} \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \leq \varepsilon.$$

Remark 4.24 *As we will also show in the proof, if $ck_0 \in \mathsf{Supp}(\mathcal{D}_{\mathcal{CK}})$, then $\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right] > 0$, so the inequality is indeed well-defined.*

Proof For every $ck_0, ck_1 \in \mathcal{CK}$, if $\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right] > 0$, then

$$\Pr_{\mathcal{P}_{ck_1}}[ck_0] := \begin{cases} \frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]}, & ck_0 \in S_{q,\varepsilon}(ck_1) \\ 0, & \text{otherwise} \end{cases},$$

whereas if $\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right] = 0$, then $\Pr_{\mathcal{P}_{ck_1}}[ck_0] := \mathbf{1}_{ck_1 = ck_0}$.

Note that for all $ck_0 \in \mathcal{CK}$ and $ck_1 \in S_{q,\varepsilon}(ck_0)$, if $ck_0 \notin \mathsf{Supp}(\mathcal{D}_{\mathcal{CK}})$, then

$$\Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] = 0; \tag{4.10}$$

otherwise, if $ck_0 \in \mathsf{Supp}(\mathcal{D}_{\mathcal{CK}})$, then since $ck_0 \in S_{q,\varepsilon}(ck_1)$ by Lemma 4.18,

$$\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right] \geq \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0] > 0,$$

and thus

$$\Pr_{\mathcal{P}_{ck_1}}[ck_0] = \frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]}. \tag{4.11}$$

Applying Lemma 4.19 then gives

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{CK},L,q} &\leq \varepsilon' + \sum_{ck_0 \in \mathcal{CK}} \max_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \\
&= \varepsilon' + \sum_{ck_0 \in \mathsf{Supp}(\mathcal{D}_{\mathcal{CK}})} \max_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] && \text{(Eq. (4.10))} \\
&= \varepsilon' + \sum_{ck_0 \in \mathsf{Supp}(\mathcal{D}_{\mathcal{CK}})} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0] \max_{ck_1 \in S_{q,\varepsilon}(ck_0)} \frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]}, && \text{(Eq. (4.11))}
\end{aligned}$$

where by applying Lemma 4.19, Eq. (4.10) and Eq. (4.11), we get

$$\begin{aligned}
\varepsilon' &= \sum_{ck_0 \in \mathcal{CK}} \sum_{ck_1 \in S_{q,\varepsilon}(ck_0)} \Pr_{\mathcal{P}_{ck_1}}[ck_0] \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \\
&= \sum_{ck_0 \in \mathsf{Supp}(\mathcal{D}_{\mathcal{CK}})} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0] \sum_{ck_1 \in S_{q,\varepsilon}(ck_0)} \frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]} \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q}. \qquad \square
\end{aligned}$$

Corollary 4.25 (Threshold-based bound) *Let C be a compressor on Σ, let $k, L \in \mathbb{N}_+$ such that $k \leq L$, and let $\mathcal{D}_{\mathcal{CK}}$ be a cookie distribution on $\mathcal{CK} \subseteq \Sigma^k$. Let $q \in \mathbb{N}$ and $0 \leq \alpha, \beta, \varepsilon \leq 1$. Let*

$$T = \left\{ck \in \mathcal{CK} : \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck] > \beta \Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck)\right]\right\}.$$

If $\Pr_{\mathcal{D}_{\mathcal{CK}}}[T] \leq \alpha$, then

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{CK},L,q} \leq \varepsilon + \alpha + \beta - \alpha\beta.$$

Proof By Theorem 4.23,

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{CK},L,q} \leq \varepsilon + \sum_{ck_0 \in \mathrm{Supp}(\mathcal{CK})} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0] \max_{ck_1 \in S_{q,\varepsilon}(ck_0)} \frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]}. \tag{4.12}$$

By the definition of T, for every $ck_0 \in \mathrm{Supp}(\mathcal{CK})$,

$$\begin{aligned}
\max_{ck_1 \in S_{q,\varepsilon}(ck_0)} \frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]} &\leq \beta + \max_{ck_1 \in S_{q,\varepsilon}(ck_0) \cap T} \left\{\frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]} - \beta\right\} \\
&\leq \beta + \sum_{ck_1 \in T} \mathbf{1}_{S_{q,\varepsilon}(ck_0)}(ck_1) \left(\frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]} - \beta\right) \\
&= \beta + \sum_{ck_1 \in T} \mathbf{1}_{S_{q,\varepsilon}(ck_1)}(ck_0) \left(\frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]} - \beta\right),
\end{aligned}$$

where the last equality is due to Lemma 4.18.

Plugging the above inequality into Eq. (4.12) gives us

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{LOCR}}_{C,\mathcal{CK},L,q} &\leq \varepsilon + \beta + \sum_{ck_0 \in \mathrm{Supp}(\mathcal{D}_{\mathcal{CK}})} \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0] \sum_{ck_1 \in T} \mathbf{1}_{S_{q,\varepsilon}(ck_1)}(ck_0) \left(\frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]} - \beta\right) \\
&= \varepsilon + \beta + \sum_{ck_1 \in T} \left(\frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]} - \beta\right) \sum_{ck_0 \in \mathrm{Supp}(\mathcal{D}_{\mathcal{CK}})} \mathbf{1}_{S_{q,\varepsilon}(ck_1)}(ck_0) \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_0] \\
&= \varepsilon + \beta + \sum_{ck_1 \in T} \left(\frac{\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1]}{\Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]} - \beta\right) \Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right] \\
&= \varepsilon + \beta + \sum_{ck_1 \in T} \left(\Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] - \beta \Pr_{\mathcal{D}_{\mathcal{CK}}}\left[S_{q,\varepsilon}(ck_1)\right]\right) \\
&\leq \varepsilon + \beta + \sum_{ck_1 \in T} (1 - \beta) \Pr_{\mathcal{D}_{\mathcal{CK}}}[ck_1] \\
&\leq \varepsilon + \beta + (1 - \beta)\alpha = \varepsilon + \alpha + \beta - \alpha\beta.
\end{aligned}$$

□

We should be able to derive similar results based on the sizes of indistinguishable sets instead of ratios. Finally, it seems possible to get more intelligible results if we relax the advantage by ε.

4.5 Bounds on simple compressors

We use Theorem 4.21 to study the LOCR security of two simple compressors, the fixed-dictionary compressor defined in [2] and Huffman coding.

4.5.1 Fixed-dictionary compressor

Construction

```
$\mathrm{FD}_{\mathsf{D},w,\ell}(x)$

1:  $y \leftarrow o$
2:  $i \leftarrow 1$
3:  $\{s_1, \ldots, s_d\} \leftarrow \mathsf{D}$
4:  while $i \leq |x| - w + 1$ do
5:    if $\exists j \in [d]$ s.t. $s_j = x[i, i + w - 1]$ then
6:      $y \leftarrow y \| $encoding of $j$
7:      $i \leftarrow i + w$
8:    else
9:      $y \leftarrow y \| $encoding of $x[i, i + \ell - 1]$
10:     $i \leftarrow i + \ell$
11: return $y$
```

Figure 4.9: The fixed-dictionary compressor $\mathrm{FD}_{\mathsf{D},w,\ell}$ defined in [2].

Figure 4.9 shows the fixed-dictionary compressor $\mathrm{FD}_{\mathsf{D},w,\ell}$ defined in [2], modified to conform to our notation. The fixed dictionary D is a set of d strings $\{s_1, \ldots, s_d\}$, each of length w. The compressor $\mathrm{FD}_{\mathsf{D},w,\ell}$ iterates through the input. At the i-th character in x, if $x[i, i + w - 1]$, the substring of length w starting from the i-th character in x, corresponds to entry s_j in the dictionary, then the compressor outputs encoding of the index j, and advances the pointer by w; otherwise, the compressor outputs the next ℓ characters $x[i, i + \ell - 1]$, and advances the pointer by ℓ.

The encoding scheme used in the fixed-dictionary compressor is only specified for a special case for evaluation, but remains generally unspecified in [2]. As the encoding scheme clearly matters for a security proof, we assume that the encoding scheme encodes every string in of length ℓ to the same length ℓ', which is consistent with the note on the decompressor pseudocode in [2].

Flaws in [2, Appendix C]

We restate the the theorem on the CR security of the fixed-dictionary compressor in [2], with necessary modifications to suit our definitions. In particular, because the corresponding security analysis in [2] does not contain any analysis of the encryption scheme, we directly translate their results to the LOCR security. We also fixed their confusion between w and ℓ.

Proposition 4.26 (Adapted from [2, Theorem 2]) *Let Ω be an alphabet, and let $d, w, \ell, n \in \mathbb{N}_+$. Let D be a set of d strings $\{s_1, \ldots, s_d\}$, each of length w. Let $\mathcal{U}_{\mathcal{CK}}$ be a uniform distribution on $\mathcal{CK} = \Omega^n$. Let $\mathcal{A}$ be a* LOCR *adversary against the fixed-dictionary compressor $\mathrm{FD}_{\mathsf{D},w,\ell}$. Then, for every $q \in \mathbb{N}$,*

$$\mathsf{Adv}^{\mathsf{LOCR}}_{\mathrm{FD}_{\mathsf{D},w,\ell},\mathcal{U}_{\mathcal{CK}},q}(\mathcal{A}) \leq 2^{-\Delta},$$

where

$$\Delta \geq \left(1 - d\left(1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-3w+1}\right)\right) \cdot \log\left(|\Omega|^{n-2w} - |\Omega|^{n-2w} \cdot d\left(1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-3w+1}\right)\right).$$

The security analysis for [2, Theorem 2] can be found in [2, Appendix C]. We highlight two notable flaws in the analysis:

- The proof of [2, Lemma 1] is incorrect. More specifically, for a fixed $x \in \Omega^w$ and $ck \leftarrow\$ \mathcal{U}_{\mathcal{CK}}$, the proof used an inequality

$$\Pr\left[\bigwedge_{i=1}^{n-w+1} x \neq ck[i, i+w-1]\right] \geq \sum_{i=1}^{n-w+1} \Pr[x \neq ck[i, i+w-1]],$$

  which was unexplained and clearly does not hold.
- In the security analysis, Δ appears to come from the conditional Shannon entropy of the secret cookie, but then $2^{-\Delta}$ cannot bound the LOCR advantage of $\mathcal{A}$, as Shannon entropy may be much larger than min-entropy, while the latter is the correct notion to use in this context.

The security analysis for [2, Theorem 2] is not easily repairable, as it relies heavily on Shannon entropy.

New proof

We provide a new security proof for the LOCR security of the fixed-dictionary compressor. The proof follows the general idea in [2, Appendix C] but uses Theorem 4.21 in place of entropy.

Theorem 4.27 *Let Ω be an alphabet, and let $d, w, \ell, n \in \mathbb{N}_+$. Let D be a set of d strings $\{s_1, \ldots, s_d\}$, each of length w. Let $\mathcal{U}_{\mathcal{CK}}$ be a uniform distribution on*

$\mathcal{CK} = \Omega^n$. Let $\mathcal{A}$ be a LOCR *adversary against the fixed-dictionary compressor* $\mathrm{FD}_{\mathsf{D},w,\ell}$*. Then, for every $q \in \mathbb{N}$,*

$$\mathsf{Adv}^{\mathsf{LOCR}}_{\mathrm{FD}_{\mathsf{D},w,\ell},\mathcal{U}_{\mathcal{CK}},q}(\mathcal{A}) \leq \frac{d(n-w+1)}{|\Omega|^w} + \frac{1}{|\Omega|^{n-2w+2}}.$$

Proof If $n - 2w + 2 \leq 0$, then the theorem trivially holds; assume hereafter that $n - 2w + 2 > 0$.

Let $x \preceq ck$ denote that x is a substring of ck.

First, we get a rough bound for $\Pr[\exists x \in \mathsf{D} : x \preceq ck]$ by applying the union bound twice:

$$\begin{aligned}\Pr[\exists x \in \mathsf{D} : x \preceq ck] &\leq \sum_{x \in \mathsf{D}} \Pr[x \preceq ck] \\ &\leq \sum_{x \in \mathsf{D}} \sum_{i=1}^{n-w+1} \Pr[x = ck[i, i+w-1]] \\ &\leq \sum_{x \in \mathsf{D}} \sum_{i=1}^{n-w+1} \frac{1}{|\Omega|^w} = \frac{d(n-w+1)}{|\Omega|^w},\end{aligned}$$

in which all probabilities are conditional to $ck \leftarrow\!\!\$\, \mathcal{U}_{\mathcal{CK}}$. Let

$$S_1 := \{ck : \exists x \in \mathsf{D} : x \preceq ck\}.$$

We proceed by using the same intuition as in [2] that a CRIME attacker can at best recover the $(w-1)$-prefix and $(w-1)$-suffix of ck. For every $ck_{\mathrm{pre}}, ck_{\mathrm{suf}} \in \Omega^{w-1}$, let $K_{ck_{\mathrm{pre}},ck_{\mathrm{suf}}} \subseteq \Omega^n$ be

$$K_{ck_{\mathrm{pre}},ck_{\mathrm{suf}}} = \left\{ck_{\mathrm{pre}} \| ck_{\mathrm{mid}} \| ck_{\mathrm{suf}} : ck_{\mathrm{mid}} \in \Omega^{n-2w+2}\right\} \setminus S_1,$$

and let S_2 be an arbitrary minimum set such that for every $ck_{\mathrm{pre}}, ck_{\mathrm{suf}} \in \Omega^{w-1}$, if $K_{ck_{\mathrm{pre}},ck_{\mathrm{suf}}} \neq \emptyset$, then $S_2 \cap K_{ck_{\mathrm{pre}},ck_{\mathrm{suf}}} \neq \emptyset$. Because S_2 is minimum, we have $|S_2| \leq |\Omega|^{2w-2}$. Let $T = S_1 \cup S_2$.

We show that for every $ck_0 \in \Omega^n$, there exists some $ck_1 \in T$ that is 0-indistinguishable (i.e. perfectly indistinguishable) from ck_0: If $\exists x \in \mathsf{D} : x \preceq ck_0$, then $ck_0 \in S_1 \subseteq T$, and we let $ck_1 = ck_0$. Otherwise, $ck_0 \notin S_1$, so there exist $ck_{\mathrm{pre}}, ck_{\mathrm{suf}} \in \Omega^{w-1}$ such that $ck_0 \in K_{ck_{\mathrm{pre}},ck_{\mathrm{suf}}}$, and by definition, there exists $ck_1 \in K_{ck_{\mathrm{pre}},ck_{\mathrm{suf}}}$ such that $ck_1 \in S_2 \subseteq T$. The cookies ck_0 and ck_1 share the same $(w-1)$-prefix and $(w-1)$-suffix and do not contain any substring in the dictionary, so by a similar argument as in [2], ck_0 and ck_1 are 0-indistinguishable if we assume the encoding scheme encodes strings of length ℓ to the same length ℓ'.

Since cookies are uniformly distributed, applying Theorem 4.21 gives

$$\begin{aligned}\mathsf{Adv}^{\mathsf{LOCR}}_{\mathsf{FD}_{\mathsf{D},w,\ell},\mathcal{U}_{\mathcal{CK}},q}(\mathcal{A}) &\leq \frac{|T|}{|\Omega|^n}\\ &\leq \frac{|S_1|}{|\Omega|^n} + \frac{|S_2|}{|\Omega|^n}\\ &\leq \Pr[\exists x \in \mathsf{D} : x \preceq ck] + \frac{1}{|\Omega|^{n-2w+2}}\\ &\leq \frac{d(n-w+1)}{|\Omega|^w} + \frac{1}{|\Omega|^{n-2w+2}}. \qquad \square\end{aligned}$$

Huffman coding

We do not specify the technical details of the Huffman coding compressor C_H here; instead, we remark that the Huffman coding compressor is what we call *insensitive to permutations*; that is, for all $x, x' \in \Sigma^+$, if x' is a permutation of x, then $|C_H(x)| = |C_H(x')|$. The reason is that Huffman coding derives prefix codes solely from the frequencies of each character in the plaintext, so the encoding of each character stays the same after permutation, and therefore the compressed length also does not change after permutation. Note that a compressor that uses a set of fixed Huffman codes are also insensitive to permutations.

Definition 4.28 *A deterministic compressor C on Σ is insensitive to permutations, if for every $\ell \in \mathbb{N}_+$ and for every $x, x' \in \Sigma^\ell$, if there exists a bijection $f : [\ell] \to [\ell]$ such that for every $i \in [\ell]$, $x[i] = x'[f(i)]$, then $|C(x)| = |C(x')|$.*

Theorem 4.29 *Let C_H be a compressor on Σ that is insensitive to permutations, let $k \in \mathbb{N}_+$, and let $\mathcal{U}_{\mathcal{CK}}$ be a uniform cookie distribution on $\mathcal{CK} = \Omega^k$, where $\Omega \subseteq \Sigma$. Then for every $q \in \mathbb{N}$,*

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C_H,\mathcal{U}_{\mathcal{CK}},q}(\mathcal{A}) \leq \left(\frac{e}{|\Omega|} + \frac{e}{k}\right)^k,$$

where $e = 2.718\ldots$ is the Euler's number.

Proof First, note that for two cookies $ck_0, ck_1 \in \Omega^k$, if ck_1 is a permutation of ck_0, then they are perfectly indistinguishable. The reason is that for all possible queries $m', m'' \in \Sigma^*$, if ck_1 is a permutation of ck_0, then $m' \| ck_1 \| m''$ is also a permutation of $m' \| ck_0 \| m''$, so $|C_H(m' \| ck_0 \| m'')| = |C_H(m' \| ck_1 \| m'')|$.

Let $T \subseteq \mathcal{CK}$ be the minimum set such that for every $ck_0 \in \mathcal{CK}$, there exists $ck_1 \in T$ such that ck_1 is a permutation of ck_0. Computing the size of T is a simple problem in combinatorics, for which the answer is $\binom{|\Omega|+k-1}{k}$.

Therefore,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{LOCR}}_{C_H,\mathcal{U}_{C\mathcal{K}},q}(\mathcal{A}) &\leq \frac{|T|}{|\Omega|^k} \\ &= \frac{1}{|\Omega|^k} \cdot \binom{|\Omega|+k-1}{k} \\ &\leq \frac{1}{|\Omega|^k}\left(e\frac{|\Omega|+k-1}{k}\right)^k \\ &\leq \left(\frac{e}{|\Omega|}+\frac{e}{k}\right)^k. \end{aligned}$$ □

4.6 Padding

4.6.1 Block padding

The most commonly used padding scheme pads a compressed message of length $\ell \in \mathbb{N}$ to length $\lceil \ell/B \rceil \cdot B$, where $B \in \mathbb{N}_+$ is the block length.

Applying a block padding scheme to the compressor output can be regarded as mapping the compressor output to a different alphabet. For example, for a compressor with an output alphabet $\{\{0,1\}^8\}^+$, applying a block padding with $B = 16$ can be seen as changing the output alphabet to $\{\{0,1\}^{128}\}^+$. For a compressor C on the input alphabet Σ, applying a block padding scheme of block length $B \in \mathbb{N}_+$ yields another compressor C_B on the input alphabet Σ but on a larger output alphabet, where for every $\ell \in \mathbb{N}_+$, $\mathcal{R}_{C_B,\ell} = \mathcal{R}_{C,\ell}$, and for every $s \in \Sigma^\ell$ and $r \in \mathcal{R}_{C,\ell}$,

$$|C_B(s;r)| = \left\lceil \frac{|C(s;r)|}{B} \right\rceil.$$

Therefore, for every $k \in \mathbb{N}_+$,

$$\begin{aligned}\Delta_k(C_B,\ell) &= \max_{r\in\mathcal{R}_{C_B,\ell}} \max_{\substack{s,s'\in\Sigma^\ell \\ \mathsf{HD}(s,s')\leq k}} \left(|C_B(s';r)| - |C_B(s;r)|\right) \\ &= \max_{r\in\mathcal{R}_{C,\ell}} \max_{\substack{s,s'\in\Sigma^\ell \\ \mathsf{HD}(s,s')\leq k}} \left(\left\lceil\frac{|C(s';r)|}{B}\right\rceil - \left\lceil\frac{|C(s;r)|}{B}\right\rceil\right) \\ &\leq \max_{r\in\mathcal{R}_{C,\ell}} \max_{\substack{s,s'\in\Sigma^\ell \\ \mathsf{HD}(s,s')\leq k}} \left\lceil\frac{|C(s';r)| - |C(s;r)|}{B}\right\rceil \\ &= \left\lceil\frac{\Delta_k(C,\ell)}{B}\right\rceil. \end{aligned}$$

where the inequality is because $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$ for all numbers x, y. We can then apply Theorem 4.10 and its corollaries to bound the LOCR security of C_B. Ideally, B should be greater or equal to $\max_{\ell \in \{k,\ldots,L\}} \Delta(C, \ell)$. Therefore, if sensitivity of the compressor C in question satisfies $\Delta(C, \ell) = \Theta(\sqrt{\ell})$, then ideally $B = \Omega(\sqrt{L})$; however, padding schemes in practice often use a small constant for B, and as a result $\Delta(C_B, \ell) = \Theta(\sqrt{\ell})$.

4.6.2 Randomised padding

Definitions

Definition 4.30 (randomised padding scheme) *A randomised padding scheme* pad *on alphabet Ω is a probabilistic distribution on Ω^* with finite support.*

Remark 4.31 *Similar to Degabriele [9], we do not consider padding schemes that depend on the message m or its length $|m|$.*[1]

Definition 4.32 (padding length distribution) *Let* pad *be a randomised padding scheme on Ω. The padding length distribution of* pad *is a distribution $\mathcal{D}_{\mathsf{pad}}$ on $\mathbb{N}$, where for every $\ell \in \mathbb{N}$,*

$$\Pr_{\mathcal{D}_{\mathsf{pad}}}[\ell] = \Pr_{\mathsf{pad}}\left[\Omega^{\ell}\right] = \sum_{s \in \Omega^{\ell}} \Pr[\mathsf{pad}() \Rightarrow s].$$

A randomised padding scheme pad *on alphabet Ω is a probabilistic distribution on Ω^* with finite support.*

Definition 4.33 (padded compressor) *Let C be a compressor on input alphabet Σ and output alphabet Ω, and let* pad *be a randomised padding scheme on Ω with padding length distribution $\mathcal{D}_{\mathsf{pad}}$.*

```
$C_{\mathsf{pad}}(m)$
1 :  $z \leftarrow\$ C(m)$
2 :  $p \leftarrow\$ \mathsf{pad}()$
3 :  return $z \| p$
```

Figure 4.10: The padded compressor C_{pad}.

The padded compressor C_{pad} constructed from C and pad *is a compressor defined in Fig. 4.10. More specifically, let $\ell \in \mathbb{N}_+$, let $\mathcal{R}_{C,\ell}$ be the set of random coins used by*

[1]Intuitively, such padding schemes may leak more information about the plaintext, but an analogy to local differential privacy hints that they may actually be better in some cases.

C on length ℓ, and let $\mathcal{R}_{\mathsf{pad}}$ be the set of random coins used by pad*; for all $m \in \Sigma^\ell$ and $r_\ell := (r_{C,\ell}, r_{\mathsf{pad}}) \in \mathcal{R}_{C,\ell} \times \mathcal{R}_{\mathsf{pad}}$,*

$$C_{\mathsf{pad}}(m; r_\ell) := C(m; r_{C,\ell}) \| \mathsf{pad}(; r_{\mathsf{pad}}).$$

Remark 4.34 *Since our subsequent discussion is only related to the padding length distribution, we do not specify a corresponding decompressor for C_{pad}. In fact, such a decompressor might not even exist for some padding length distributions.*

Bounding LOCI with difference hiding

Similar to [9, Theorem 3.3], we relegate the LOCI game on a padded compressor to distinguishing between samples from pad() or $d + \mathsf{pad}()$.

```
Game BD-HIDE($\mathcal{A}, \mathcal{D}_{\mathsf{pad}}, \delta$)
1 : $b \leftarrow\$ \{0,1\}$
2 : $b' \leftarrow\$ \mathcal{A}^{\mathsf{Hide}}()$
3 : return $b = b'$
```

```
Oracle Hide($d$)
1 : if $|d| > \delta$ then
2 :     return $\perp$
3 : $p \leftarrow\$ \mathcal{D}_{\mathsf{pad}}$
4 : return $b \cdot d + p$
```

Figure 4.11: Game BD-HIDE for $\mathcal{D}_{\mathsf{pad}}$ and $\delta \in \mathbb{N}$, a bidirectional version of the D-HIDE game in [9]. We use a different formulation to bypass a restriction in Degabriele's formulation; namely, the Left-or-Right oracle in Degabriele's length hiding game requires that $|m_0| \le |m_1|$.

Consider the game BD-HIDE in Fig. 4.11, in which the oracle Hide takes an integer d that is possibly negative. The advantage of $\mathcal{A}$ in game BD-HIDE is

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta}(\mathcal{A}) = |2\Pr[G(\mathcal{A}, \mathcal{D}_{\mathsf{pad}}, \delta) \Rightarrow \mathsf{true}] - 1|.$$

The BD-HIDE security with regard to $\mathcal{D}_{\mathsf{pad}}$, $\delta \in \mathbb{N}$ and $q \in \mathbb{N}$ is

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta,q} = \sup\{\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta}(\mathcal{A}) : \forall \mathcal{A} \text{ s.t. } \mathcal{A} \text{ makes at most } q \text{ queries to } \mathsf{Hide}\}.$$

Theorem 4.35 *Let C be a compressor on input alphabet Σ and output alphabet Ω, and* pad *be a randomised padding scheme on Ω with padding length distribution $\mathcal{D}_{\mathsf{pad}}$; let C_{pad} be a padded compressor constructed from C and* pad.

Let $k, L \in \mathbb{N}_+$ such that $k \le L$, $\mathcal{CK} \subseteq \Sigma^k$, $ck_0, ck_1 \in \mathcal{CK}$, and $q \in \mathbb{N}$. Then

$$\mathsf{Adv}^{\mathsf{LOCI}}_{C_{\mathsf{pad}},ck_0,ck_1,L,q} \le \min\left\{\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q}, \mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta_{k,L},q}\right\},$$

where

$$\delta_{k,L} = \max_{\ell \in \{k,\ldots,L\}} \Delta_{\mathsf{HD}(ck_0,ck_1)}(C, \ell).$$

The proof is similar to Degabriele's proof sketch of [9, Theorem 3.3], except that our proof is under a slightly different setting.

Proof See Appendix B.2. □

Bounding difference hiding with distance measures

A natural next step is to attempt to transform a BD-HIDE adversary into a non-adaptive adversary. Intuitively, an adversary can just query Hide with the maximum possible d and therefore does not need to be adaptive. However, on a closer look, it appears possible for an adaptive adversary to perform strictly better than any non-adaptive adversary, even if the distribution is simple. This barrier may partly explain the gap between the D-HIDE game and the multisample distinguisher theorem (Theorem 3.10) in [9].

We first give a weaker linear bound in this thesis, which can also be proved via a standard hybrid argument.

Definition 4.36 *A padding length distribution $\mathcal{D}_{\mathsf{pad}}$ is unimodal if for every $x_1, x_2, x_3 \in \mathbb{N}$ where $x_1 < x_2 < x_3$, we have $\Pr_{\mathcal{D}_{\mathsf{pad}}}[x_2] \geq \min(\Pr_{\mathcal{D}_{\mathsf{pad}}}[x_1], \Pr_{\mathcal{D}_{\mathsf{pad}}}[x_3])$.*

Remark 4.37 *All padding length distributions considered in [9] are unimodal.*

Theorem 4.38 *Let $\mathcal{D}_{\mathsf{pad}}$ be a padding length distribution, $\delta, q \in \mathbb{N}$, $\mathbf{x}$ be a random variable distributed according to $\mathcal{D}_{\mathsf{pad}}$. Then*

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta,q} \leq q \cdot \max_{d \in [\delta]} \mathsf{SD}\left(\mathbf{x}, \mathbf{x} + d\right).$$

In particular, if $\mathcal{D}_{\mathsf{pad}}$ is unimodal, then

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta,q} \leq q \cdot \mathsf{SD}\left(\mathbf{x}, \mathbf{x} + \delta\right),$$

Proof For simplicity, we define all random variables that appear in the proof on the image $\mathbb{Z}$. Let $\mathcal{B}$ be a BD-HIDE adversary against $\mathcal{D}_{\mathsf{pad}}$ and δ that makes at most q queries. Without loss of generality, we assume that $\mathcal{B}$ is deterministic and makes exactly q queries, and for each query $\mathsf{Hide}(d)$, we have $|d| \in \{-\delta, -\delta + 1, \ldots, \delta\}$.

In game BD-HIDE with $\mathcal{B}$, for every $i \in [q]$, let $\mathbf{p}_i$ be a random variable that denotes the sample from $\mathcal{D}_{\mathsf{pad}}$ when answering the i-th query, let $\mathbf{d}_i$ be a random variable that denotes the i-th query of $\mathcal{B}$ to Hide when $b = 1$, and let $\mathbf{h}_i = b \cdot \mathbf{d}_i + \mathbf{p}_i$ be the answer to the i-th query. Since we assumed that $\mathcal{B}$ is deterministic, $\mathbf{d}_i$ can be written as a function of $\mathbf{h}_1, \ldots, \mathbf{h}_{i-1}$, and the output of $\mathcal{B}$ can be written as a function $f_{\mathcal{B}}(\mathbf{h}_1, \ldots, \mathbf{h}_q)$. Therefore,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta}(\mathcal{B}) &= \left|\Pr\left[f_{\mathcal{B}}(\mathbf{h}_1, \ldots, \mathbf{h}_q) = 0 \mid b = 0\right] - \Pr\left[f_{\mathcal{B}}(\mathbf{h}_1, \ldots, \mathbf{h}_q) = 0 \mid b = 1\right]\right| \\ &\leq \mathsf{SD}\left((\mathbf{p}_i)_{i \in [q]}, (\mathbf{d}_i + \mathbf{p}_i)_{i \in [q]}\right).\end{aligned}$$

By triangle inequality,

$$\begin{aligned}
&\mathsf{SD}\left((\mathbf{p}_i)_{i\in[q]}, (\mathbf{d}_i+\mathbf{p}_i)_{i\in[q]}\right)\\
&\le \sum_{t=1}^{q} \mathsf{SD}\left((\mathbf{d}_1+\mathbf{p}_1,\dots,\mathbf{d}_{t-1}+\mathbf{p}_{t-1},\mathbf{p}_t),(\mathbf{d}_1+\mathbf{p}_1,\dots,\mathbf{d}_t+\mathbf{p}_t)\right)\\
&\le \sum_{i=1}^{q} \max_{d\in\{-\delta,\dots,\delta\}} \mathsf{SD}\left(\mathbf{p}_t, d+\mathbf{p}_t\right)\\
&= q\cdot \max_{d\in\{-\delta,\dots,\delta\}} \mathsf{SD}\left(\mathbf{x}, d+\mathbf{x}\right).
\end{aligned}$$

Note that $\mathsf{SD}(\mathbf{x},\mathbf{x}) = 0$, and for every $d\in[\delta]$,

$$\mathsf{SD}(\mathbf{x}, d+\mathbf{x}) = \mathsf{SD}(\mathbf{x}-d, \mathbf{x}) = \mathsf{SD}(\mathbf{x}, -d+\mathbf{x}),$$

so

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta}(\mathcal{B}) \le q\cdot \max_{d\in\{-\delta,\dots,\delta\}} \mathsf{SD}(\mathbf{x}, d+\mathbf{x}) = q\cdot \max_{d\in[\delta]} \mathsf{SD}(\mathbf{x}, d+\mathbf{x}).$$

We now discuss the case where $\mathcal{D}_{\mathsf{pad}}$ is unimodal. For every $d\in[\delta-1]$ and $x\in\mathbb{N}$, if $\Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d] < \Pr_{\mathcal{D}_{\mathsf{pad}}}[x]$, then $\Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d] \ge \Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d+1]$; otherwise, we have $\Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d] < \min\{\Pr_{\mathcal{D}_{\mathsf{pad}}}[x], \Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d+1]\}$, contradictory to the unimodality of $\mathcal{D}_{\mathsf{pad}}$. Therefore, for every $d\in[\delta-1]$,

$$\begin{aligned}
\mathsf{SD}(\mathbf{x}, d+\mathbf{x}) &= \sum_{\substack{x\in\mathbb{N}\\ \Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d]<\Pr_{\mathcal{D}_{\mathsf{pad}}}[x]}} \left(\Pr_{\mathcal{D}_{\mathsf{pad}}}[x] - \Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d]\right)\\
&\le \sum_{\substack{x\in\mathbb{N}\\ \Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d]<\Pr_{\mathcal{D}_{\mathsf{pad}}}[x]}} \left(\Pr_{\mathcal{D}_{\mathsf{pad}}}[x] - \Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d+1]\right)\\
&\le \sum_{\substack{x\in\mathbb{N}\\ \Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d+1]<\Pr_{\mathcal{D}_{\mathsf{pad}}}[x]}} \left(\Pr_{\mathcal{D}_{\mathsf{pad}}}[x] - \Pr_{\mathcal{D}_{\mathsf{pad}}}[x+d+1]\right)\\
&= \mathsf{SD}(\mathbf{x}, d+1+\mathbf{x}).
\end{aligned}$$

Thus, if $\mathcal{D}_{\mathsf{pad}}$ is unimodal, then

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta}(\mathcal{B}) \le \max_{d\in[\delta]} \mathsf{SD}(\mathbf{x}, d+\mathbf{x}) \le \mathsf{SD}(\mathbf{x}, \delta+\mathbf{x}). \qquad \square$$

Theorem 4.38 is already sufficient for studying uniform padding schemes. However, it fails to capture Degabriele's motivation for choosing Gaussian padding over uniform padding when q is large.

Then, we derive a similar bound to the multisample distinguisher theorem (Section 3.2.6). Our bound also applies to adaptive adversaries, bridging the gap in [9]. Our proof is derived from the intermediate results in the Chi-squared method by Dai et al. [8] and a lemma for the Squared-Ratio method by Chen et al. [7]. We note that the bound can likely be further optimised, which we leave for future work.

Lemma 4.39 (adapted from [7, Lemma 1]) *Let P and Q be two distributions over the discrete set Γ. For any subset $\Gamma' \subset \Gamma$ such that $\Gamma' \subseteq \mathrm{Supp}(Q)$, one has*

$$\sum_{x\in\Gamma'} |\Pr_P[x] - \Pr_Q[x]| \leq \left(2\sum_{x\in\Gamma'} \Pr_P[x] \ln\left(\frac{\Pr_P[x]}{\Pr_Q[x]}\right) + 2\sum_{x\in\Gamma\setminus\Gamma'} \Pr_P[x] - \Pr_Q[x]\right)^{\frac{1}{2}}.$$

Remark 4.40 *The lemma is a generalisation of Pinsker's inequality (Lemma 3.4).*

Theorem 4.41 *Let $\mathcal{D}_{\mathsf{pad}}$ be a padding length distribution, $\delta, q \in \mathbb{N}$. Let $T = \{-\delta, -\delta+1, \ldots, \delta\}$. Let*

$$W = 1 - \min_{d\in T} \sum_{\substack{x\geq d \\ x\in\mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})}} \Pr_{\mathcal{D}_{\mathsf{pad}}}[x-d],$$

and

$$K = \max_{d\in T} \sum_{\substack{x\geq d \\ x\in\mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})}} \Pr_{\mathcal{D}_{\mathsf{pad}}}[x-d] \ln\left(\frac{\Pr_{\mathcal{D}_{\mathsf{pad}}}[x-d]}{\Pr_{\mathcal{D}_{\mathsf{pad}}}[x]}\right).$$

Then

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta,q} \leq \frac{qW}{2} + \sqrt{\frac{q(K+W)}{2}}.$$

Remark 4.42 *For a* BD-HIDE *adversary $\mathcal{B}_{na}$ that always queries the oracle* Hide *with the same value as the first query it makes, applying the multisample distinguisher theorem (Theorem 3.10) gives*

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta,q}(\mathcal{B}_{na}) \leq 2qW + \sqrt{\frac{qK}{2(1-W)}}.$$

Proof For simplicity, we define all random variables that appear in the proof on the image $\mathbb{Z}$. Let $\mathbf{x}$ be a random variable distributed according to $\mathcal{D}_{\mathsf{pad}}$. Let $\mathcal{B}$ be a BD-HIDE adversary against $\mathcal{D}_{\mathsf{pad}}$ and δ that makes at most q queries. Without loss of generality, we assume that $\mathcal{B}$ is deterministic and makes exactly q queries, and for each query $\mathsf{Hide}(d)$, we have $|d| \in T$.

In game BD-HIDE with $\mathcal{B}$, for each $i \in [q]$, let $\mathbf{p}_i$ be a random variable that denotes the sample from $\mathcal{D}_{\mathsf{pad}}$ when answering the i-th query, and let $\mathbf{d}_i$ be a random variable that denotes the i-th query of $\mathcal{B}$ to Hide when $b = 1$. As in the proof for Theorem 4.38, we have

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta}(\mathcal{B}) \leq \mathsf{SD}\left((\mathbf{p}_i)_{i\in[q]}, (\mathbf{d}_i + \mathbf{p}_i)_{i\in[q]}\right). \tag{4.13}$$

We have

$$\begin{aligned}
&\sum_{(x_1,\dots,x_q)\in\mathbb{Z}^q\setminus\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})^q} \Pr\left[\bigwedge_{i=1}^{q} \mathbf{d}_i+\mathbf{p}_i=x_i\right]\\
&=1-\sum_{(x_1,\dots,x_q)\in\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})^q} \Pr\left[\bigwedge_{i=1}^{q} \mathbf{d}_i+\mathbf{p}_i=x_i\right]\\
&=\Pr\left[\bigvee_{i=1}^{q} \mathbf{d}_i+\mathbf{p}_i\notin\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})\right]\\
&\le\sum_{i=1}^{q}\Pr[\mathbf{d}_i+\mathbf{p}_i\notin\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})].
\end{aligned}$$

For every $i\in[q]$, because $\mathbf{d}_i$ and $\mathbf{p}_i$ are independent,

$$\begin{aligned}
\sum_{i=1}^{q}\Pr[\mathbf{d}_i+\mathbf{p}_i\notin\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})] &= \sum_{i=1}^{q}\sum_{d\in T}\Pr[\mathbf{d}_i=d]\Pr[\mathbf{p}_i+d\notin\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})]\\
&\le\sum_{i=1}^{q}\max_{d\in T}\Pr[\mathbf{p}_i+d\notin\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})]\\
&=q\cdot\max_{d\in T}\Pr[\mathbf{x}+d\notin\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})].
\end{aligned}$$

Therefore,

$$\sum_{(x_1,\dots,x_q)\in\mathbb{Z}^q\setminus\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})^q} \Pr\left[\bigwedge_{i=1}^{q} \mathbf{d}_i+\mathbf{p}_i=x_i\right]\le q\cdot W. \tag{4.14}$$

We can follow the derivation of [8, Eq. (5)] to apply Lemma 4.39 and get

$$\begin{aligned}
&\sum_{x_1,\dots,x_q}\frac{1}{\sqrt{2}}\left|\Pr\left[\bigwedge_{i=1}^{q}\mathbf{d}_i+\mathbf{p}_i=x_i\right]-\Pr\left[\bigwedge_{i=1}^{q}\mathbf{p}_i=x_i\right]\right|\\
&\le\left(\sum_{x_1,\dots,x_q}\Pr\left[\bigwedge_{i=1}^{q}\mathbf{d}_i+\mathbf{p}_i=x_i\right]\ln\left(\frac{\Pr\left[\bigwedge_{i=1}^{q}\mathbf{d}_i+\mathbf{p}_i=x_i\right]}{\Pr\left[\bigwedge_{i=1}^{q}\mathbf{p}_i=x_i\right]}\right)+q\cdot W\right)^{\frac{1}{2}}\\
&=\left(\sum_{x_1,\dots,x_q}\sum_{t=1}^{q}\Pr\left[\bigwedge_{i=1}^{q}\mathbf{d}_i+\mathbf{p}_i=x_i\right]\ln\left(\frac{\Pr\left[\mathbf{d}_t+\mathbf{p}_t=x_t\;\middle|\;\bigwedge_{i=1}^{t-1}\mathbf{d}_i+\mathbf{p}_i=x_i\right]}{\Pr[\mathbf{x}=x_t]}\right)+q\cdot W\right)^{\frac{1}{2}}\\
&=\left(\sum_{t=1}^{q}\sum_{x_1,\dots,x_t}\Pr\left[\mathbf{d}_t+\mathbf{p}_t=x_t\;\middle|\;\bigwedge_{i=1}^{t-1}\mathbf{d}_i+\mathbf{p}_i=x_i\right]\ln\left(\frac{\Pr\left[\mathbf{d}_t+\mathbf{p}_t=x_t\;\middle|\;\bigwedge_{i=1}^{t-1}\mathbf{d}_i+\mathbf{p}_i=x_i\right]}{\Pr[\mathbf{x}=x_t]}\right)+q\cdot W\right)^{\frac{1}{2}},
\end{aligned}$$

in which $x_1,\dots,x_q$ each takes values from $\operatorname{Supp}(\mathcal{D}_{\mathsf{pad}})$.

Because for every $t \in [q]$, $\mathbf{d}_t$ is a function of $(\mathbf{d}_1 + \mathbf{p}_1, \dots, \mathbf{d}_{t-1} + \mathbf{p}_{t-1})$, for every $(x_1, \dots, x_{t-1}) \in \mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})^{t-1}$, we have

$$\begin{aligned}
&\sum_{x_t \in \mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})} \Pr\left[\mathbf{d}_t + \mathbf{p}_t = x_t \;\middle|\; \bigwedge_{i=1}^{t-1} \mathbf{d}_i + \mathbf{p}_i = x_i\right] \ln\left(\frac{\Pr\left[\mathbf{d}_t + \mathbf{p}_t = x_t \;\middle|\; \bigwedge_{i=1}^{t-1} \mathbf{d}_i + \mathbf{p}_i = x_i\right]}{\Pr[\mathbf{x} = x_t]}\right) \\
&\le \max_{d \in T} \sum_{x_t \in \mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})} \Pr[\mathbf{p}_t = x_t - d] \ln\left(\frac{\Pr[\mathbf{p}_t = x_t - d]}{\Pr[\mathbf{x} = x_t]}\right) \\
&= \max_{d \in T} \sum_{x_t \in \mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})} \Pr[\mathbf{x} = x_t - d] \ln\left(\frac{\Pr[\mathbf{x} = x_t - d]}{\Pr[\mathbf{x} = x_t]}\right).
\end{aligned}$$

So

$$\begin{aligned}
&\sum_{t=1}^{q} \sum_{x_1, \dots, x_t} \Pr\left[\mathbf{d}_t + \mathbf{p}_t = x_t \;\middle|\; \bigwedge_{i=1}^{t-1} \mathbf{d}_i + \mathbf{p}_i = x_i\right] \ln\left(\frac{\Pr\left[\mathbf{d}_t + \mathbf{p}_t = x_t \;\middle|\; \bigwedge_{i=1}^{t-1} \mathbf{d}_i + \mathbf{p}_i = x_i\right]}{\Pr[\mathbf{x} = x_t]}\right) \\
&\le \sum_{t=1}^{q} \sum_{x_1, \dots, x_{t-1}} \Pr\left[\bigwedge_{i=1}^{t-1} \mathbf{d}_i + \mathbf{p}_i = x_i\right] \max_{d \in T} \sum_{x_t \in \mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})} \Pr[\mathbf{x} = x_t - d] \ln\left(\frac{\Pr[\mathbf{x} = x_t - d]}{\Pr[\mathbf{x} = x_t]}\right) \\
&\le q \cdot \max_{d \in T} \sum_{x \in \mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})} \Pr[\mathbf{x} = x - d] \ln\left(\frac{\Pr[\mathbf{x} = x - d]}{\Pr[\mathbf{x} = x]}\right) = q \cdot K, \qquad (4.15)
\end{aligned}$$

in which $x_1, \dots, x_t$ each takes values from $\mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})$. Then

$$\sum_{x_1, \dots, x_q \in \mathrm{Supp}(\mathcal{D}_{\mathsf{pad}})} \frac{1}{2} \left| \Pr\left[\bigwedge_{i=1}^{q} \mathbf{d}_i + \mathbf{p}_i = x_i\right] - \Pr\left[\bigwedge_{i=1}^{q} \mathbf{p}_i = x_i\right] \right| \le \sqrt{\frac{q(K+W)}{2}}. \qquad (4.16)$$

Combining Eqs. (4.13) to (4.16), we get

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}}, \delta}(\mathcal{B}) &\le \mathsf{SD}\left((\mathbf{p}_i)_{i \in [q]}, (\mathbf{d}_i + \mathbf{p}_i)_{i \in [q]}\right) \\
&\le \frac{qW}{2} + \sqrt{\frac{q(K+W)}{2}},
\end{aligned}$$

and the proof is complete. □

Bounding LOCR with difference hiding

Theorem 4.43 *Let C be a compressor,* pad *be a randomised padding scheme with a padding length distribution $\mathcal{D}_{\mathsf{pad}}$, and C_{pad} be a padded compressor constructed from C and* pad. *Let $h, k, L \in \mathbb{N}_+$ such that $h \le k \le L$, Σ be an alphabet of size $\sigma > 1$, $\mathcal{U}_{\mathcal{CK}}$ be a uniform cookie distribution on $\mathcal{CK} = \Sigma^k$, and $q \in \mathbb{N}$. Then*

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C_{\mathsf{pad}}, \mathcal{U}_{\mathcal{CK}}, L, q} \le \min_{h \in [k]} \left\{ \mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}}, \delta_{k,L,h}, q} + \frac{1}{\sum_{i=0}^{h} \binom{k}{i} (\sigma - 1)^i} \right\},$$

where for every $h \in [k]$,

$$\delta_{k,L,h} := \max_{\ell \in \{k,\dots,L\}} \Delta_h(C, \ell).$$

Proof By Theorem 4.35. for every $ck_0, ck_1 \in \mathcal{CK}$,

$$\mathsf{Adv}^{\mathsf{LOCI}}_{C_{\mathsf{pad}}, ck_0, ck_1, L, q} \leq \mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}}, \delta_{k,L,\mathsf{HD}(ck_0,ck_1)}, q}.$$

Note also that for every $\delta_1, \delta_2 \in \mathbb{N}$ where $\delta_1 \leq \delta_2$,

$$\mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}}, \delta_1, q} \leq \mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}}, \delta_2, q},$$

because a BD-HIDE adversary against $\mathcal{D}_{\mathsf{pad}}$ and δ_1 is naturally also a BD-HIDE adversary against $\mathcal{D}_{\mathsf{pad}}$ and δ_2.

Let h be an arbitrary element in $[k]$, and let $\varepsilon = \mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}}, h, q}$. For every cookie $ck_0 \in \mathcal{CK}$, consider $S_{ck_0} := \{ck_1 \in \mathcal{CK} : \mathsf{HD}(ck_0, ck_1) \leq h\}$; we have

$$|S_{ck_0}| = \sum_{i=0}^{h} \binom{k}{i} (\sigma - 1)^i,$$

and $S_{ck_0} \subseteq S_{q,\varepsilon}(ck_0)$. Therefore, applying Corollary 4.25 with $\alpha = 0$ and $\beta = 1/(\sum_{i=0}^{h} \binom{k}{i} (\sigma-1)^i)$ gives

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C_{\mathsf{pad}}, \mathcal{U}_{\mathcal{CK}}, L, q} \leq \min_{h \in [k]} \left\{ \mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}}, \delta_{k,L,h}, q} + \frac{1}{\sum_{i=0}^{h} \binom{k}{i} (\sigma - 1)^i} \right\}, \qquad \square$$

Corollary 4.44 *Let C be a compressor, $B \in \mathbb{N}$, pad be a randomised padding scheme with a uniform padding length distribution $\mathcal{D}_{\mathsf{pad}}$ on $\{0, \dots, B\}$, and C_{pad} be a padded compressor constructed from C and pad. Let $h, k, L \in \mathbb{N}_+$ such that $h \leq k \leq L$, Σ be an alphabet of size $\sigma > 1$, $\mathcal{U}_{\mathcal{CK}}$ be a uniform cookie distribution on $\mathcal{CK} = \Sigma^k$, and $q \in \mathbb{N}$. Then*

$$\mathsf{Adv}^{\mathsf{LOCR}}_{C_{\mathsf{pad}}, \mathcal{U}_{\mathcal{CK}}, L, q} \leq \min_{h \in [k]} \left\{ q \frac{\delta_{k,L,h}}{B+1} + \frac{1}{\sum_{i=0}^{h} \binom{k}{i} (\sigma - 1)^i} \right\},$$

where for every $h \in [k]$,

$$\delta_{k,L,h} := \max_{\ell \in \{k,\dots,L\}} \Delta_h(C, \ell).$$

More generally, we can apply Theorems 4.38 and 4.41 to study other randomised padding schemes, such as Gaussian padding and Laplace padding. However, as Degabriele noted [9], we do not have a closed form of expression for the distance measures of discrete Gaussian padding schemes, and must rely on approximation or numerical evaluation, which we leave for future work and refer readers to [9] for more details.

Finally, we note that the evaluation of the CRIME attack in [9] does not apply to all compression side-channel attacks, because in the CRIME attack considered in [9], the compressed lengths are assumed to only differ by one depending on whether the guess for the next byte is correct. This assumption does not hold for Kelsey's attacks (Section 3.5.1), the looking ahead technique in BREACH [19], as well as our amplification techniques in Chapter 5. The attacker in [9] also does not perform backtracking.

On the contrary, selecting parameters for Gaussian padding according to its effect on the CRIME attack may make Gaussian padding more vulnerable to our amplification techniques in Chapter 5. Intuitively, if the compressed lengths can have a larger difference than the average padding length, then the bounds in Theorems 3.10 and 4.41 may favour uniform padding than Gaussian padding. We leave the evaluation of compression side-channel attacks against different randomised padding schemes for future work.

Chapter 5

Amplification on DEFLATE/zlib

In this chapter, we present novel techniques to reduce or remove the noise in compression side-channel attacks against DEFLATE and zlib by amplifying differences in compressed lengths with specially crafted queries.

5.1 Attack model and assumptions

In this section, we present a model for our attacks in Chapters 5 and 6, and highlight two special assumptions to our techniques.

5.1.1 Attack model

Our attack model can be seen as an abstraction of the "CRIME-style" attacks in e.g. [19,48].

Let Σ and Ω be two alphabets such that $\Omega \subseteq \Sigma \subseteq \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$. Let $\ell, \ell_p, L_q \in \mathbb{N}_+$. A target possesses a secret cookie $ck \in \Omega^\ell$, and the goal of an attacker is to recover ck through a compression side channel.

We abstract the compression side channel as a compression length oracle, to which an attacker can submit queries $Q \in \Sigma^*$, where $|Q| \leq L_q$. The oracle responds to each query Q with the compressed length ℓ_C of a plaintext embedded with both the secret and the query; namely

$$\ell_C = |C(\ldots \| P \| ck \| \ldots \| Q \| \ldots)|,$$

where C represents a compressor that implements the DEFLATE algorithm described in Section 3.3, or, when specified, the zlib compressor (Section 3.4), and $P \in \Sigma^{\ell_p}$ is a prefix. The attacker is assumed to know the compressor C and the prefix P. Both the prefix P and the secret ck stay unchanged among different queries, but the omitted parts in the plaintext might be unknown to the attacker and might change with each query. The attacker can adaptively query the compression length oracle.

We require that $\ell_p \geq \mathsf{min\_match} - 1$, where $\mathsf{min\_match}$ is the minimum match length. We also require that Q is close to ck in the plaintext, and in particular their distance is smaller than the sliding window size $\mathsf{max\_window}$.

Our model captures a subset of adversaries playing in the length-only cookie recovery (LOCR) game (Section 4.2.1) against DEFLATE or zlib, in that an adversary in our model can only control a part of the plaintext located after the secret.

It is also possible to make certain variations to our model without invalidating our techniques; for example, an attacker may know a suffix to the secret instead of a prefix, or there could be a format restriction on the queries.

A toy example

We use the following example setting for illustration in Chapters 5 and 6. Let Σ be the set of printable ASCII characters without the symbol `&`, and let $\Omega \subseteq \Sigma$ be the set of hexadecimal characters, i.e. $\Omega = \{\texttt{0}, \ldots, \texttt{9}, \texttt{a}, \ldots, \texttt{f}\}$. The target's secret $ck \in \Omega^\ell$ is a random value represented in the hexadecimal format. The target exposes an compression length oracle to an attacker, in which the target uses zlib with default parameters to compress a string

$$\texttt{...\&secret=}ck\texttt{\&...\&query=}Q\texttt{\&...}$$

for each query $Q \in \Sigma^*$ of length $|Q| \leq L_q$, and returns the length of the compressed string to the attacker. The attacker knows the compression level used by the target as well as the prefix to the secret `secret=`, and tries to recover the secret by making adaptive queries to the oracle.

5.1.2 Special assumptions

In addition to the assumptions in Section 5.1.1 that are shared by most compression side-channel attacks, our new techniques in Chapters 5 and 6 generally rely on two special assumptions:

1. The secret ck is located before a query Q, but not necessarily directly; i.e. the index of ck is smaller than the index of Q in the plaintext.
2. The maximum length of a query L_q is not too small.

The first assumption is present in our attack model, but is generally not required for existing compression side-channel attacks; in fact, the 16K-1 technique in [48] only works if the secret is located after the query. Unfortunately, this assumption is crucial for almost all of our techniques.

Nevertheless, we expect that it should be feasible for a query to be located after the secret in compression side channels. Indeed, many existing compression side-channel attacks were based on such queries, e.g. [21, 41]. For web servers vulnerable to BREACH and its variants, an example where this assumption holds was given in [38]. While CRIME exploited the request path located before the secret cookie in HTTP requests, it is known to be possible to launch a similar attack with the control of another cookie located after the secret cookie.

The second assumption on the maximum length of a query L_q is also important, since longer queries are intuitively more powerful, which is also suggested by our analysis on the effect of compressor sensitivity on compression side-channel attacks in Chapter 4. In this chapter, L_q determines the amplification that the attacker could eventually get. We will quantify the effect of L_q on each of our technique in the following sections.

Several existing techniques for compression side-channel attacks already used long queries: the 16K-1 technique in CRIME [48], as the name suggests, utilised queries of around 16KiB; in the divide-and-conquer technique, the length of a query can reach hundreds of bytes when the cookie alphabet is large; DBREACH [21] required filling a 4KiB page with attacker-controlled data to compute the compressibility of a guess.

Some of our techniques may also rely on other assumptions, which we will discuss separately in the following sections.

5.2 Overview of amplification

5.2.1 Motivation

Compression side-channel attacks are known to be sensitive to noise, as they rely on length differences of at most a few bytes in the compressed data. However, random data are often present in the plaintext to be compressed, and a target may also introduce noise to the compressor as a heuristic defence against compression side-channel attacks (Section 2.3.4). Performing more adaptive queries can help mitigate noise, but at the cost of making attacks less practical and easier to detect.

As a concrete example, we consider a simple compression side-channel attack in our example setting. In order to check if $ck[1] = \texttt{a}$, an attacker may make a query `secret=a`. if the first character in ck is indeed `a`, then LZ77 replaces the whole query `secret=a` with a back-reference to its previous occurrence; otherwise, LZ77 only replaces `secret=` with a back-reference and keeps the character `a` in the query as is. Therefore, the lengths of the LZ77 outputs differ by one depending on whether $ck[1] = \texttt{a}$, and the length difference likely persists in the compressed data after Huffman coding, from which the attacker can learn if $ck[1] = \texttt{a}$.

In the above example, the length difference depending on whether $ck[1] = \texttt{a}$ can hardly exceed more than a few bytes, because the (i) LZ77 output is virtually the same between the two cases, and (ii) the Huffman codes are either fixed or largely determined by other parts of the plaintext. However, the contraposition of the above arguments suggests that, if we want to amplify the compressed length difference, then we might first try to design queries for which either (i) or (ii) does not hold. Because LZ77 adapts to its input, it appears possible to design queries for which the LZ77 outputs can be very different depending on the secret, which we will exploit for amplification.

5.2.2 General idea

We amplify the compression side channel by designing queries to make the LZ77 outputs differ substantially depending on the secret, in ways such that the differences in LZ77 outputs can be translated into differences in compressed lengths. Because LZ77 adapts to the input it has scanned, finding such queries is surprisingly easy and can be done in several ways. We highlight three mechanisms in DEFLATE/zlib that we exploit:

- The LZ77 algorithm itself;
- The combination of Huffman coding and LZ77;
- The LZ77 hash table in zlib.

We describe our amplification techniques for the setting where an attacker wants to check whether $ck[1]$ is equal to a guess $g \in \Omega$, which will be generalised in Chapter 6. Note that our techniques naturally generalise to checking the next unknown character in ck: if an attacker has already recovered the first $\ell' < \ell$ characters of the secret ck, then by letting $P' = P[1, \ell']$ and $ck' = ck[\ell' + 1, \ell]$, the problem of checking whether $ck[\ell' + 1]$ is equal to g is reduced to checking whether $ck'[1]$ is equal to g.

We loosely define the *amplification ratio* to measure the efficiency of our amplification techniques, which is the absolute value of the ratio between the expected compressed length differences depending on whether $ck[1] = g$ and the length of the query $|Q|$. We refrain from giving a formal definition as this would require specifying the distributions of ck and the unspecified parts in the plaintext, but we note that the amplification ratio is upper bounded by the sensitivity of the compressor in question if we assume the unspecified parts in the plaintext are independent from ck.

5.2.3 Query structure

The queries we design follow the general structure we describe below, which we will also reuse in Chapter 6.

A query may contain up to four parts, each represented as a string in Σ^*. They include two dictionary strings dict_1 and dict_2, an optional filler string filler between the two dictionary strings, and a string of gadgets $\mathsf{gadgets}$. Generally,

$$Q = \mathsf{dict}_1 \| \mathsf{filler} \| \mathsf{dict}_2 \| \mathsf{gadgets}$$

It is not necessary for dict_1, filler, dict_2 and $\mathsf{gadgets}$ be next to each other, as long as they follow this order in the query. Only $\mathsf{gadgets}$ is required to be located after the secret in the plaintext, while dict_1, filler and dict_2 are allowed to appear before the secret, possibly in separate parts of the plaintext.

The dictionary strings dict_1 and dict_2 each represent a set of strings called a dictionary. One way to construct a dictionary string dict from a set of strings $\{s_1, \ldots, s_d\}$ is

$$\mathsf{dict} = \mathsf{sep} \| s_1 \| \mathsf{sep} \| s_2 \| \mathsf{sep} \| \ldots \| \mathsf{sep} \| s_d \| \mathsf{sep}$$

where the separator sep is preferably a character that never or only rarely appears in other parts of the plaintext, while dict_1 and dict_2 can share the same separator sep. For example, we can write a dictionary string for the set $\{\texttt{foo}, \texttt{bar}\}$ as `$foo$bar$`.

The filler string filler is optional and may be omitted with possible performance degradation. The purpose of filler is to increase the distance between the dictionary strings dict_1 and dict_2. For example, a filler string can be a single character repeated multiple times or a randomly sampled string.

The string of gadgets $\mathsf{gadgets}$ is the concatenation of one or more specially crafted strings, each being an individual gadget. Each gadget exploits the DEFLATE algorithm to perform a desired functionality. We will elaborate on the concept of gadgets and present more types of gadgets in Chapter 6.

In this chapter, we do not present our amplification techniques from the perspective of gadgets, but we note that all constructions for $\mathsf{gadgets}$ in this chapter except telescoping (Section 5.3) can be seen as the concatenation of a byte-match gadget (Section 6.3.1) and an amplify gadget (Section 6.3.4).

5.3 Telescoping

Telescoping is a simple but convenient technique for amplification. This technique achieves a compressed length difference of several bytes with relatively short queries, which makes it particularly useful in cases where the maximum query length L_q is small. Telescoping can also be used in concert with other amplification techniques as a "free lunch".

Akagi et al. have used similar techniques to derive lower bounds on the sensitivities of several LZ-family compressors [1]. However, to our knowledge, there are no existing works that used telescoping or similar techniques for compression side-channel attacks.

Telescoping does not require dict_1, filler or dict_2 and only needs $\mathsf{gadgets}$. Suppose an attacker wishes to check whether the character $ck[1]$ immediately following a known prefix P of length ℓ_p is equal to a guess $g \in \Omega$. The attacker can choose $k \in \mathbb{N}_+$ where $k \leq \ell_p + 2 - \mathsf{min\_match}$, and query

$$Q = P[k, \ell_p] \| g \| \mathsf{sep}_1 \| P[k-1, \ell_p] \| g \| \mathsf{sep}_2 \| \dots \| \mathsf{sep}_{k-1} \| P[1, \ell_p] \| g$$

which can be

`et=1Aret=1Bcret=1Cecret=1Dsecret=1`

in our example setting when the guess is `1` and $k = \ell_p - 2 = 5$. Note that, if $k = 1$, then Q degenerates to $P \| g$, or to `secret=1` in our example.

The separators $\mathsf{sep}_1, \dots, \mathsf{sep}_{k-1}$ are pairwise distinct strings such that their first characters $\mathsf{sep}_1[1], \dots, \mathsf{sep}_{k-1}[1]$ are preferably not in Ω. It usually suffices to use separators of one or two characters each, but longer separators also work, and in practice, the substrings $P[i, \ell_p] \| g$ may be in different parts of the plaintext, in which case the separators can be seen as the unspecified plaintexts in between. The separators $\mathsf{sep}_1, \dots, \mathsf{sep}_{k-1}$ can also be the same when lazy matching is disabled.

In order for the telescoping technique to work with high probability, one should additionally ensure that:

1. The substring $P \| ck[1]$ can be found by LZ77 in the sliding window before the query Q;
2. Besides in the known prefix P, the substring $P[k, \ell_p]$ has no or few occurrences in the sliding window before the query Q.

If $g = ck[1]$, then LZ77 likely transforms the query Q into

$$\texttt{<L}(\ell_p - k + 2)\texttt{,D*>} \| \mathsf{sep}_1 \| \texttt{<L}(\ell_p - k + 3)\texttt{,D*>} \| \mathsf{sep}_2 \| \dots \| \mathsf{sep}_{k-1} \| \texttt{<L}(\ell_p + 1)\texttt{,D*>}$$

Otherwise, if $g \neq ck[1]$, then LZ77 likely transforms Q into

$$\texttt{<L}(\ell_p - k + 1)\texttt{,D*>} \| g \| \mathsf{sep}_1 \| \texttt{<L}(\ell_p - k + 2)\texttt{,D*>} \| g \| \mathsf{sep}_2 \| \dots \| \mathsf{sep}_{k-1} \| \texttt{<L}\ell_p\texttt{,D*>} \| g$$

Thus, after Huffman coding, the compressed length when $g = ck[1]$ is likely smaller than when $g \neq ck[1]$ by around k bytes. However, the exact length difference depends on Huffman coding: for example, if k is large, then the character g may be encoded more efficiently by Huffman coding, making the length difference smaller. The amplification ratio for telescoping is roughly $(\ell_p - k/2)^{-1}$, and the maximum compressed length difference is roughly ℓ_p. Therefore, telescoping can only increase the compressed length difference by several bytes if the known prefix is short.

Finally, we note that reversing the direction of telescoping, such as querying `secret=1Aecret=1Bcret=1...` in our example, only works correctly in zlib if lazy matching is disabled and $\ell_p - k + 2 >$ max_insert. Otherwise, LZ77 is likely to match the substring $P[k, \ell_p] \| g$ in the query to its previous occurrence in the query $P[k-1, \ell_p] \| g$ regardless of the actual value of $ck[1]$. The same problem applies for every substring $P[i, \ell_p] \| g$ whose previous occurrences in the query are inserted into the hash table in LZ77.

5.4 Chaining

In this section, we describe several techniques for amplification that we refer to as *chaining*.

We motivate the chaining techniques by making the following observation. When an attacker queries $P \| g$, depending on whether $ck[1] = g$, LZ77 not only produces different outputs, but also moves its pointer to different positions in the plaintext. If $ck[1] = g$, then LZ77 likely replaces the whole query with a single back-reference, and moves the pointer just past the query; otherwise, LZ77 likely only replaces P in the query with a back-reference, and moves the pointer to the position of g.

If the plaintext after $P \| g$ is not adversarially chosen, then when $ck[1] \neq g$, the LZ77 pointer at g likely moves just by one in the next step and coincides with the pointer position when $ck[1] = g$. However, with carefully designed strings following the query, it is possible to maintain the small difference in pointer positions and utilise the difference to create diverging LZ77 outputs.

5.4.1 Setup

Suppose an attacker wishes to check whether the character $ck[1]$ immediately following a known prefix P of length ℓ_p is equal to a guess $g \in \Omega$.

Let $m, k \in \mathbb{N}_+$, where $k \geq$ min_match. Here m indicates the number of segments in the chain, and k indicates the length of each segment. A larger m increases the compressed length difference, but it also increases the query size and the probability of failure. A larger k means that the chain is more likely to work as desired, but it also decreases the amplification ratio and thus makes the query less space-efficient.

Let S be a string of length $\ell_s \in \mathbb{N}_+$ chosen by the attacker, where ℓ_s is a function of m and k to be specified later. The character $S[1]$ preferably does not take values in Ω, and the substring $S[2, \ell_s]$ can be a random string of length $\ell_s - 1$. The string S should not contain special characters we designate for other purposes, such as the separator for dict_1 and dict_2. For example, one can construct $S[2, \ell_s]$ by sampling from alphanumeric characters.

A chaining query consists of four parts dict_1, filler, dict_2 and $\mathsf{gadgets}$ as specified in Section 5.2.3. The dictionary strings dict_1 and dict_2 are to be specified later; the filler string filler is optional and is constructed as described in Section 5.2.3. The string of gadgets $\mathsf{gadgets}$ is of the form $P\|g\|S$. That is, the query is of the form

$$\mathsf{dict}_1\|\mathsf{filler}\|\mathsf{dict}_2\|P\|g\|S$$

We assume that the substring $P\|ck[1]$ can be found by LZ77 in the sliding window before Q, but $P[2, \ell_p]gS[1]$ does not not appear in the sliding window. Therefore, when the LZ77 pointer is at $P[1]$ in the query, if $ck[0] = g$, then LZ77 finds a match for $P\|g$ in the sliding window, and moves the pointer to $S[1]$ in the query; otherwise, if $ck[0] \neq g$, then the LZ77 pointer moves to the position of g in the query at some point.

To simplify our discussion, we abuse the notation to let $S[0]$ denote the character before S in the plaintext (i.e. g), and let $S[\ell_s + 1]S[\ell_s + 2]$ denote the characters that immediately follow S in the plaintext, or end symbols if there are no more characters available.

5.4.2 Distance-based chain

A distance-based chain amplifies the compressed length difference by exploiting the interaction of LZ77 and Huffman coding in DEFLATE. More specifically, depending on whether the guess is correct, LZ77 encodes the string of gadgets $\mathsf{gadgets}$ in a distance-based chain into two different sequences of back-references, such that the sequence of back-references when the guess is correct tend to be encoded more efficiently.

Recall that, for a back-reference `<L`x`,D`y`>` of length x and distance y, the Huffman coding process in DEFLATE maps the distance y to a code between 0 and 29 and some extra bits, with the extra bits stay unchanged thereafter; see Section 3.3.2. The number of extra codes required increases as y grows; for example, a distance of 50 requires 4 extra bits to encode, while a distance of 500 requires 7 extra bits (see Table 3.1).

We describe the construction of a distance-based chain based on Section 5.4.1. Let the length of the string S be $\ell_s = mk$. The dictionary string dict_1 represents a set of strings $\{S[(i-1)k, ik-1]\}_{i \in [m]}$, and dict_2 represents a set of strings $\{S[(i-1)k+1, ik]\}_{i \in [m]}$.

In our example setting, suppose the guess is `1`, $m = 4$ and $k = 3$. Then the query could be

```
-1AB-CDE-FGH-IJK------------ABC-DEF-GHI-JKL-secret=1ABCDEFGHIJKL
```

For the distance-based chain to work correctly, we should ensure that:

1. For each $i \in [m]$, $S[(i-1)k, ik-1]$ in dict_1 can be found by LZ77 in the sliding window before Q, but $S[(i-1)k, ik]$ does not appear in the sliding window.
2. For each $i \in [m]$, $S[(i-1)k+1, ik]$ in dict_2 can be found by LZ77 in the sliding window before Q, but $S[(i-1)k+1, ik+1]$ and $S[(i-1)k+2, ik+2]$ do not appear in the sliding window.

The above conditions are likely to be met if one uses random bytes to construct the gadget, but other constructions are also possible.

If $ck[1] = g$, then LZ77 finds a match for $P\|g$ and moves the pointer to $S[1]$ in the query Q. Next, by condition 2, LZ77 likely replaces $S[1,k], \ldots, S[(m-1)k+1, mk]$ with back-references to their respective occurrences in dict_2, and the pointer moves just past S. Otherwise, $ck[1] \neq g$, then LZ77 finds a match for P and moves the pointer to g in the query Q. Next, by conditions 1 and 2, LZ77 likely replaces $g\|S[1,k-1], \ldots, S[(m-1)k, mk-1]$ by back-references to their respective occurrences in dict_2, and the pointer moves to $S[mk]$. That is, the LZ77 algorithm can be seen to group gadgets as

$$\begin{aligned} &P\|g, S[1,k], \ldots, S[(m-1)k+1, mk] && (ck[1] = g) \\ &P, g\|S[1,k-1], \ldots, S[(m-1)k, mk-1], S[mk] && (ck[1] \neq g) \end{aligned}$$

While the length of the LZ77 output differ by only one depending on whether $ck[1] = g$, the back-reference distances are larger when $ck[1] \neq g$ as most of them refer to substrings in the farther dictionary dict_1, and therefore may need more extra bits to encode. Because the extra bits stay unchanged in Huffman coding, if we assume Huffman coding compresses other parts of LZ77 outputs to similar lengths, then the differences in the number of extra bits persist to affect the difference of compressed lengths, making the compressed length larger when $ck[1] \neq g$.

The Huffman coding should be stable and only affect the amplification minimally. That said, it appears possible to construct adversarial examples in which the Huffman coding partially or completely offsets the length change from extra bits. In practice, this probability is negligible and in particular much lower than the probability that some prerequisites of the distance-based chain are not satisfied.

It is challenging to determine the amplification ratio for a distance-based chain; as a rough estimation, if each back-reference to dict_1 needs two more extra bits to encode its distance than a back-reference to dict_2, then the gadget creates a compressed length difference of around $m/4$ bytes with around $(3k+2)m$ bytes, yielding an amplification ratio of around $1/12k$. The length of the query is bounded by L_q as well as the sliding window size $\mathsf{max\_window}$.

We remark that one can immediately obtain a distance-based chain that amplifies compressed length differences in the reverse direction, i.e. increase the compressed length when $ck[1] = g$, by switching dict_1 and dict_2.

5.4.3 Match-based chain

A match-based chain amplifies compressed length differences by making LZ77 find different matches to gadgets that have different lengths.

We describe the constructions of match-based chains based on Section 5.4.1. It suffices to use one dictionary string for our techniques to work, but having two dictionary strings can slightly improve performance in ways similar to distance-based chains.

To give readers the general idea of a match-based chain, we first describe in Section 5.4.3 how to construct a match-based chain when lazy matching is disabled. Then, we discuss in Section 5.4.3 the more interesting and simultaneously more challenging case where lazy matching is enabled.

Lazy matching disabled

Let $k \geq \mathsf{min\_match} + 1$, and let the length of S be $\ell_s = mk$. The dictionary string dict_1 represents a set of strings $\{S[(i-1)k, ik-2]\}_{i\in[m]}$, and dict_2 represents a set of strings $\{S[(i-1)k+1, ik]\}_{i\in[m]}$.

In our example setting, suppose the guess is `1`, $m = 3$ and $k = 4$. Then the query could be

```
-1AB-DEF-HIJ----ABCD-EFGH-IJKL-secret=1ABCDEFGHIJKL
```

If $ck[1]$ is indeed `1`, then LZ77 likely groups gadgets as `secret=1`, `ABCD`, `EFGH`, `IJKL`; otherwise, LZ77 likely groups gadgets as `secret=`, `1AB`, `C`, `DEF`, `G`, `HIJ`. However, in the latter case, if lazy matching is enabled, then LZ77 can improve `1AB` with the lazy match `ABCD`, and the subsequent LZ77 output is the same regardless of $ck[1]$.

More formally, if $ck[1] = g$, then LZ77 likely groups gadgets as

$$P\|g, S[1,k], \ldots, S[(m-1)k+1, mk]$$

and outputs

$$\texttt{<L}(\ell_p+1)\texttt{,D*>}\|\texttt{<L}k\texttt{,D*>}\|\ldots\|\texttt{<L}k\texttt{,D*>}$$

Otherwise, $ck[1] \neq g$, then LZ77 likely groups gadgets as

$$P, g\|S[1,k-2], S[k-1], \ldots, S[(m-1)k, mk-2], S[mk-1], S[mk]$$

and outputs

$$\texttt{<L}\ell_p\texttt{,D*>}\|\texttt{<L}(k-1)\texttt{,D*>}\|S[k-1]\|\ldots\|\texttt{<L}(k-1)\texttt{,D*>}\|S[mk-1]\|S[mk]$$

Lazy matching enabled

We first remark that when lazy matching is enabled, the above construction for the case where lazy matching is disabled still works correctly in zlib when $k \geq \text{max\_lazy}+1$, which is useful for level 4 in zlib.

Let $k \geq \text{min\_match}+2$. In zlib, we also require that $k \leq \text{max\_lazy}+1$. Let the length of S be $\ell_s = mk-1$. The dictionary string dict_1 represents a set of strings

$$\{S[(i-1)k+2, ik]\}_{i\in[m-1]} \cup \{S[(m-1)k+1, mk-2]\},$$

and dict_2 represents a set of strings

$$\{g\|S[1,k-2]\} \cup \{S[ik-1,(i+1)k-2]\}_{i\in[m-1]}.$$

In our example setting, suppose the guess is `1`, $m=3$ and $k=5$. Then the query could be

```
-BCDE-GHIJ-KLMN----1ABC-DEFGH-IJKLM-secret=1ABCDEFGHIJKLMN
```

If $ck[1]$ is indeed `1`, then in gadgets, LZ77 first finds a match for `secret=1` and moves the pointer to `A`. There, LZ77 finds a match `ABC`, but subsequently improves it with the lazy match `BCDE`, and moves the pointer to `F`. LZ77 finds a match `FGH` at `F`, but then improves it with the lazy match `GHIJ`, after which LZ77 finds another match `KLMN`. In summary, when $ck[1]=g$, then LZ77 likely groups gadgets as `secret=1`, `A`, `BCDE`, `F`, `GHIJ`, `KLMN`. Otherwise, if $ck[1]\neq g$, then LZ77 likely groups gadgets as `secret=`, `1ABC`, `DEFGH`, `IJKLM`.

More formally, if $ck[1]=g$, then LZ77 likely groups gadgets as

$$P\|g, S[1], S[2,k], \ldots, S[(m-2)k+1], S[(m-2)k+2,(m-1)k], S[(m-1)k+1, mk-1]$$

and outputs

$$\texttt{<L}(\ell_p+1)\texttt{,D*>}\|S[1]\|\texttt{<L}(k-1)\texttt{,D*>}\|\ldots\|S[(m-2)k+1]\|\texttt{<L}(k-1)\texttt{,D*>}\|\texttt{<L}(k-1)\texttt{,D*>}$$

Otherwise, $ck[1] \neq g$, then LZ77 likely groups gadgets as

$$P, g \| S[1, k-2], S[k-1, 2k-2] \ldots, S[(m-1)k-1, mk-2], S[mk-1]$$

and outputs

$$\texttt{<L}\ell_p\texttt{,D*>} \| \texttt{<L}(k-1)\texttt{,D*>} \| S[k-1] \| \ldots \| \texttt{<L}k\texttt{,D*>} \| S[mk-1] \| S[mk]$$

Note our construction for the final part of the chain can clearly be further optimised. For example, we can remove the string $S[(m-1)k+1, mk-2]$ (`KLMN` in our example) from dict_1 to get an additional compressed length difference of around one. However, as we will see in Chapter 6, our design allows for a modular treatment of all amplification chains.

Note also that the amplification chain constructed above *increases* the compressed length when $ck[1] = g$.

Estimated performance

If we make a simplifying assumption that Huffman coding encodes each back-reference to about the same length in both cases, then the compressed length difference is around m. The length of the query is about $(3k+1)m$, so the amplification ratio is approximately $1/3k$. The length of the query is bounded by L_q as well as the sliding window size $\mathsf{max\_window}$.

5.4.4 Optimisations

Reducing dictionary size

We briefly discuss several ways to reduce the query size in the chaining technique by using shorter dictionary strings.

A natural strategy is to choose the string S such dict_1 or dict_2 represents a set of strings with fewer than m entries, and is therefore shorter than the normal length of around $(k+1)m$. For example, in a distance-based chain, dict_2 represents $\{S[(i-1)k+1, ik]\}_{i\in[m]}$; if we choose S such that for every $i' \in [\lfloor m/2 \rfloor]$, we have

$$S[(2i'-1)k+1, 2i'k] = S[(2i'-2)k+1, (2i'-1)k]$$

then dict_1 represents a set with at most $\lceil m/2 \rceil$ entries. In our example setting, when $g = 1$, a distance-based chain with $m = 4$ and $k = 3$ could be

```
-1AB-CAB-CDE-FDE------------ABC-DEF-secret=1ABCABCDEFDEF
```

An attacker may also choose S such that dict_1 or dict_2 can be encoded more efficiently. In our example setting, when $g = 1$, a possible distance-based chain with $m = 4$ and $k = 3$ is

```
-1DE-FCD-EBC-DAB------------ABCDEF-secret=1DEFCDEBCDABC
```

Finally, if an attacker knows some parts of the plaintext located before the query, then they may use the known substrings as a dictionary string to construct the query, reducing the size of $\mathtt{dict}_1$.

Match-based chains can also be optimised in similar ways.

We remark that all of the above optimisation strategies require S to be structured, so care should be taken to ensure that the correctness of the chain still holds with high probability.

Quick check

One way to check whether an amplification chain is correct is to test it in a simulated attack environment, and see if it produces the desired results. That is, an attacker could embed the query into plaintexts they constructed to the best of their knowledge, and see if the compressed lengths indeed change as desired depending on whether the guess is the same as the secret character in the plaintext simulated by the attacker.

If the check succeeds, then the attacker submits the query to the oracle, which is likely to produce desired results in the real attack environment; otherwise, if the check fails, then the attacker generates another query with fresh randomness and repeats the above process, until a preset number of attempts has been reached.

5.5 Collision-based amplification

In this section, we sketch a technique that we refer to as *collision-based amplification*, which amplifies compressed length differences in zlib by creating collisions in its LZ77 hash table implementation.

Note that the collision-based amplification techniques we describe in this section do not target the general DEFLATE algorithm, because they depend on details of the hash table implementation that are unspecified in DEFLATE, and in particular the (non-cryptographic) hash function used. That said, our techniques likely also apply to other DEFLATE compressors with similar hash table implementations as zlib, and we expect most DEFLATE compressors to be susceptible to variants of collision-based amplification.

The general idea for collision-based amplification is very similar to chaining, in that both try to make LZ77 find very different back-references depending on whether a guess is correct. In collision-based amplification, a query contains some adversarially chosen substrings with the same hash value, such that the corresponding hash chain becomes too long and the older entries on that chain are no longer visible to LZ77. As a result, LZ77 cannot find a match at certain positions in the query despite a matching substring is present in the sliding window, but can find a match at slightly different positions, creating a divergence in LZ77 outputs.

There are many possible ways to design queries for collision-based amplification. Our technique mainly utilises strings with a special property, but, as the readers will notice, the use of such strings is mainly for efficiency, and more general techniques for collision-based amplification certainly exist.

For simplicity, we focus on the default parameters for all compression levels in zlib where $\mathsf{min\_match} = 3$ and $\mathsf{hash\_bits} = 15$, but our techniques should also work similarly for other reasonable parameters.

Our presentation is again based on the following scenario: Suppose an attacker wishes to check whether the character $ck[1]$ immediately following a known prefix P of length ℓ_p is equal to a guess $g \in \Omega$.

5.5.1 Collisions

Hash collisions

Recall that in the hash table implementation of zlib, the hash value of a 3-byte string abc is

$$\mathsf{hash}(a, b, c) = \left(a \cdot 2^{10} + b \cdot 2^5 + c\right) \mod 2^{15},$$

where each byte is treated as an unsigned integer.

Notice a simple fact that the number of collisions for each hash value is balanced: for each possible hash value in $\{0, \ldots, 2^{15} - 1\}$, the number of 3-byte strings with that hash value is precisely $(2^8)^3/2^{15} = 2^9$. However, if we limit the alphabet to alphanumeric characters, then the collisions naturally become unbalanced, with the highest number of collisions being 27, for which we give an example in Table 5.1.

`0pq`	`0qQ`	`0r1`	`1Pq`	`1QQ`	`1R1`	`20q`	`21Q`	`221`
`Ppq`	`PqQ`	`Pr1`	`QPq`	`QQQ`	`QR1`	`R0q`	`R1Q`	`R21`
`ppq`	`pqQ`	`pr1`	`qPq`	`qQQ`	`qR1`	`r0q`	`r1Q`	`r21`

Table 5.1: The 27 alphanumeric strings with the hash value 20081.

Doubly-colliding strings

We will mainly consider all strings T with what we call the *doubly-colliding* property with regard to a given byte z; that is, let $abcd = T[1, 4]$, then

$$\mathsf{hash}(a, b, c) = \mathsf{hash}(b, c, d) = \mathsf{hash}(z, z, z).$$

If hash is a random function, then one would expect that such strings are extremely rare for any given z. However, because zlib uses a "rolling" hash function, if $\mathsf{hash}(a,b,c) = \mathsf{hash}(z,z,z)$, then for each possible value of d,

$$\begin{aligned}\mathsf{hash}(b,c,d) &= \left(\mathsf{hash}(a,b,c)\cdot 2^5 + d\right) \mod 2^{15}\\ &= \left(\mathsf{hash}(z,z,z)\cdot 2^5 + d\right) \mod 2^{15}\\ &= \left(z\cdot 2^{10} + z\cdot 2^5 + d\right) \mod 2^{15},\end{aligned}$$

so $\mathsf{hash}(b,c,d) = \mathsf{hash}(z,z,z)$ if and only if $d = z$.

5.5.2 Infinite chain

Up to now, all of our chaining techniques require adding more dictionary entries as m grows. However, it is possible to design a collision-based amplification chain that continues infinitely with some bootstrap data before the chain. The basic idea is to leverage two sequences of doubly-colliding strings with regard to some byte z, $Z_1 = (T_{1,i})_{i\in[k]}$, $Z_2 = (T_{2,i})_{i\in[k]}$, where one sequence is the permutation of the other one. The string of gadgets $\mathsf{gadgets}$ is of the form

$$P\|(S_1\|S_2)^m$$

where $k = \lfloor \mathsf{max\_chain}/4 \rfloor$,

$$\begin{aligned}S_1 &= g\|T_{1,1}\|\dots\|T_{1,k}\|R\\ S_2 &= g\|T_{2,1}\|\dots\|T_{2,k}\|R\end{aligned}$$

and R contains z repeated by a number of times.

To bootstrap the chain, we add S_1, S_2 to dict_1, add $\{T_{1,i}\}_{i\in[k]}$ to dict_2, and set filler to $z^{\mathsf{max\_chain}}$. We design S_1 and S_2 such that when the LZ77 pointer is at $T_{2,i}$ in S_2, then it finds the corresponding entry $T_{1,i'}$ in S_1, but not to the previous occurrence of S_2, and similarly for S_1. Therefore, when $ck = g$, we force LZ77 to take shorter matches, but when $ck \neq g$, starting at a different position without a collision, LZ77 can first find long matches in dict_1, and later from the previous occurrences of S_1 and S_2.

We have built an example for compression level 5, where we choose $k = 8$,

$$\begin{aligned}Z_1 &= (\texttt{ppqQ, pr1Q, qPqQ, qR1Q, r0qQ, r21Q, PpqQ, Pr1Q}),\\ Z_2 &= (\texttt{qR1Q, qPqQ, pr1Q, ppqQ, r21Q, r0qQ, Pr1Q, PpqQ}),\\ R &= \texttt{QQQQ}\end{aligned}$$

where the amplification ratio is around 0.185 as m grows. However, we did not test for other levels. We expect that such examples can be found for most compression levels, possibly with longer doubly-colliding strings or variations, which we leave for future work.

Chapter 6

Query programming on DEFLATE/zlib

In this chapter, we introduce a new paradigm called *query programming* for designing complex queries in compression side-channel attacks against DEFLATE and zlib.

6.1 Overview

We provide an overview of query programming in this section. The attack model and assumptions for query programming can be found in Section 5.1.

6.1.1 Motivation and general idea

As the reader may have noticed in Chapter 5, most of our amplification techniques create queries whose string of gadgets gadgets consists of two parts: the first part $P\|g$ checks whether g is the correct guess for $ck[1]$, and the second part S amplifies the compressed length difference. It is therefore natural to treat each part as an individual *gadget* that provides a certain functionality. In our case, the two gadgets are a byte-match gadget (Section 6.3.1) and an amplify gadget (Section 6.3.4).

The byte-match gadget $P\|g$ creates a difference in the LZ77 pointer positions depending on whether $ck[1] = g$, while the amplify gadget maintains that difference and leverages the difference to make LZ77 find very different back-references. More generally, a gadget transitions one set of LZ77 pointer positions to another, possibly bringing differences in LZ77 outputs at different LZ77 pointer positions.

A core observation that motivates query programming is that we can see LZ77 pointer positions as *states*, and abuse the LZ77 algorithm in DEFLATE for a controlled state transition. This observation leads to a modular way to design gadgets and make them work in concert to perform a complex functionality in what we call a *(CRIME) program*. In particular, by utilising logical gadgets, it is possible to combine multiple queries into a single one, debunking a myth that a query can only leak around one bit of information in compression side-channel attacks.

6.1.2 Structure

The query structure for query programming is the same as in Section 5.2.3. Recall that a query Q contains up to four parts: two dictionary strings dict_1 and dict_2, each representing a set of strings called a dictionary, an optional filler string filler, and a string of gadgets $\mathsf{gadgets}$; that is,

$$Q = \mathsf{dict}_1 \| \mathsf{filler} \| \mathsf{dict}_2 \| \mathsf{gadgets}$$

The string $\mathsf{gadgets}$ is the concatenation of a series of specifically crafted strings, each being a gadget that performs a specified functionality.

We refer to Q as a (CRIME) program. For simplicity, we often only describe the composition of $\mathsf{gadgets}$ when we specify a program, because the entries in dict_1 and dict_2 are determined by $\mathsf{gadgets}$.

6.1.3 Execution and output

The output of a program Q is the answer to the the query Q given by the compression oracle, i.e. the compressed length of the plaintext with Q embedded after the secret. We refer to the process of compressing the plaintext with DEFLATE/zlib as a program execution.

6.1.4 State

The state in a program execution is indicated by the LZ77 pointer position when LZ77 scans through $\mathsf{gadgets}$. Without loss of generality, we denote the state where the LZ77 pointer is at the last character of a gadget as false, and denote the state where the LZ77 pointer is at the first character of a gadget (i.e. aligned with the gadget) as true. All other pointer positions have undefined states. Note that every gadget should have at least two characters for the states to be well-defined.

We note that it is clearly possible but likely impractical in general to use ternary or quaternary states.

6.1.5 Gadget

A gadget takes an initial state and transitions it into another state. The initial state for a gadget is the state when the LZ77 pointer falls within the gadget or at the character before the gadget for the first time. The state transition derives the initial state for the next gadget, if there is any.

In the following sections, we denote a gadget by a string S of length $\ell_s \geq 2$. We abuse the notation to let $S[0]$ denote the character before the gadget in the plaintext, and let $S[\ell_s + 1]S[\ell_s + 2]$ denote the characters that immediately follow S in the plaintext, or end symbols if there are no more characters available.

6.2 Separation gadgets

We first describe two types of simple gadgets used to separate gadgets from other parts of the query, or to separate two gadgets.

6.2.1 Align gadget

An align gadget forces the LZ77 pointer that falls within the gadget to move just past it, to the first character of the next gadget. More formally, an align gadget transitions any initial state, which can be `true`, `false` or undefined, to the state `true` for the next gadget.

The motivation for align gadgets is that, in most cases, for a gadget to perform its intended functionality, the LZ77 pointer must be either aligned with the gadget or just before the gadget. In our example setting, an attacker may query `secret=1` to check if $ck[1]$ is equal to `1`. The query field is then `query=secret=1`. However, if the plaintext before the query contains a substring `memory=secret`, then LZ77 likely outputs `que<L9,D*>=1` regardless of the guess, causing the attack to fail. To solve this problem, the attacker could prepend an align gadget to the query to separate `secret=1` from `query`, such that LZ77 can find a match for `secret=1`.

We list three candidates for the align gadget; other constructions are also possible.

- *Repeated characters.* The repetition of some character that ideally does not appear elsewhere, such as `@@@@@@`, could work as an align gadget.
- *Known match.* One could use a known substring in the sliding window as an align gadget. In the above example, the attacker may use `memory` or `secret` as an align gadget, but `memory=` does not work.
- *Random string.* A random string, such as `HJx0c6`, could also work as an align gadget. One could also repeat the random string to obtain a known match.

Care must be taken when using multiple align gadgets to avoid interference among them.

Note that, the align gadget, and more generally the idea of ensuring the initial pointer position for each gadget, may not be necessary for real-world attacks, which generally allow some level of inaccuracy. Nevertheless, in the compression side-channel attack against Threema [41, Appendix B], the former of the two required canary strings possibly acted as an align gadget.

6.2.2 Dummy gadget

Sometimes, concatenating two gadgets directly may create a substring that affect the correctness of the program; a dummy gadget acts as a separator between two interfering gadgets by transitioning an initial state of true to true, and an initial state of false to false.

We give two possible constructions for a dummy gadget S of length ℓ_s:

- *Random string.* We could select S as a random string, and add both $S[0, \ell_s - 1]$ and $S[1, \ell_s]$ to arbitrary dictionaries.
- *Known match.* We could also select S to be a known substring of length ℓ_S in the sliding window, and add $S[0, \ell_s - 1]$ to an arbitrary dictionary.

Note that a dummy gadget can be seen as a degenerate case of a distance-based amplify gadget (Section 6.3.4) when $m = 1$. Conversely, an amplify gadget also provides the functionality of a dummy gadget.

6.3 Match and amplify gadgets

6.3.1 Byte-match gadget

A byte-match gadget checks whether $P\|g$ can be found the sliding window for a string P of length ℓ_p and a character g, where $\ell_p \geq \mathsf{min\_match} - 1$. Here P does not have to appear in the sliding window for the gadget to work correctly.

If the initial state is true, then the byte-match gadget changes the state to true if $P\|g$ can be found in the sliding window, and changes the state to false otherwise. The behaviour of the byte-match gadget is unspecified when the initial state is false or undefined.

A byte-match gadget can simply be $P\|g$. If $\ell_p \geq \mathsf{min\_match}$ and LZ77 might not find P in the sliding window, then we can add P to a dictionary. Adding P ensures that the program returns false when only a proper suffix of $P\|g$ but not $P\|g$ itself can be found in the sliding window.

In our example, a byte-match gadget could be `secret=1` for the prefix `secret` and guess `1`.

To ensure the correctness of the byte-match gadget, we require that $S[2, \ell_s + 1]$ does not appear in the sliding window; recall that $S[\ell_s + 1]$ is the character following the gadget and can be chosen by the attacker.

It is easy to see why the byte-match gadget works correctly. If LZ77 can find $P \| g$ in the sliding window, then LZ77 finds a match for the whole gadget, which by our assumption cannot be improved by lazy matching. Therefore, LZ77 outputs a back-reference for the whole gadget and moves its pointer to $S[\ell_s + 1]$, which represents the state true. Otherwise, LZ77 either only matches P in the query or slides through it if $\ell_p <$ min_match, and moves the pointer to $S[\ell_s]$, which represents the state false.

6.3.2 Lazy-match gadget

It is also possible to design match gadgets based on lazy matching in LZ77. A lazy-match gadget checks whether a string T can be found in the sliding window, where $|T| \geq$ min_match. In zlib, we also require $|T| \leq$ max_lazy.

If the initial state is true, then the lazy-match gadget changes the state to false if T can be found in the sliding window, and changes the state to true otherwise. The behaviour of the lazy-match gadget is unspecified when the initial state is false or undefined, or when lazy matching is disabled.

Note that, contrary to the byte-match gadget, the lazy-match gadget changes the state to true if T *cannot* be found sliding window. If a program uses both types of match gadgets together, then the NOT gadget described in Section 6.4.1 may be helpful.

The lazy-match gadget is of the form $S = T \| c$ of length $\ell_s = |T| + 1$, where c is a character that ideally only appears in lazy-match gadgets. In addition, we add $S[2, \ell_s]$ to one of the two dictionaries.

In our example, to probe whether `secret=123` is in the sliding window, a lazy-match gadget could be `secret=123/`, and we add `ecret=123/` to dict$_2$.

To ensure the correctness of the lazy-match gadget, we require that the substring $S[2, \ell_s]$ does not appear in the sliding window.

If LZ77 can find T in the sliding window, then LZ77 finds a match for $S[1, \ell_s - 1]$ in the gadget, which cannot be improved by lazy matching. Therefore, LZ77 moves the pointer to $S[\ell_s]$, which represents false. Otherwise, LZ77 either finds a match shorter than T, to be replaced by the lazy match $S[2, \ell_s]$ of length $|T|$, or does not find any match at $S[1]$ and proceeds to find the match $S[2, \ell_s]$ at $S[2]$; in both cases, the pointer moves to $S[\ell_s]$, which represents the state true.

The lazy-match gadget is essentially the same as byte-match gadget. However, the lazy-match gadget should be used with care, because it always adds a non-random entry to a dictionary, which may interfere with other gadgets, and lazy matching is only performed under certain conditions.

6.3.3 Telescoped-match gadget

The telescoping technique in Section 5.3 can be seen as a single match gadget that also decreases the compressed length by several bytes when $P\|g$ can be found in the sliding window, which we call a telescoped-match gadget. The telescoped-match gadget likely works when regardless of the initial state.

Like for the byte-match gadget (Section 6.3.1), if LZ77 might not find P in the sliding window, then we can add P to one of the two dictionaries.

We remark that a telescoped-match gadget can be seen as a compact variant of a program with k byte-match gadgets, separated by align gadgets.

6.3.4 Amplify gadgets

An amplify gadget increases the compressed length difference between the initial states true and false, and keep the initial state for the next gadget. The constructions for amplify gadgets are already covered implicitly when we introduce our amplification techniques, and can be easily extracted from our description in Chapter 5.

Depending on the direction of amplification, there could be two types of amplification gadgets: one type increases the compressed length when the initial state is true, and the other type increases the compressed length when the initial state is false. Note that one type of amplification gadgets can be transformed into the other by prepending and appending NOT gadgets (Section 6.4.1).

6.4 Logical gadgets

We have already seen that align gadgets (eventually) set the state to true, dummy gadgets merely keep the initial state, match gadgets set different states based on the match result when the initial state is true, and amplify gadgets change the compressed length difference based on the initial state while keeping that state.

In this section, we will introduce some logical gadgets to equip us with more control over states.

6.4.1 NOT gadget

As the name suggests, a NOT gadget inverts the state of the previous gadget. That is, if the initial state is true, then the NOT gadget changes the state to false; otherwise, if the initial state if false, then the NOT gadget changes the state to true.

We give two possible ways to build a NOT gadget S:

- The gadget can be a string S with length $\ell_s = \mathsf{min\_match} - 1$, such that LZ77 cannot find a match for $S[1, \ell_s + 1]$ in the sliding window. We add $S[0, \ell_s]$ to a dictionary if LZ77 might not find a match for $S[0, \ell_s]$ in the sliding window. The string S ideally contains a special character reserved for NOT gadgets at one position, and generally random strings or known matches can also work.

  In our example, to invert the state after the gadget `secret=1`, we could use a NOT gadget `!0`, and add `1!0` to the dictionary dict_2. If the initial state for the NOT gadget is true, then the pointer is first at `!`, and then advances the pointer by one byte to `0`, which represents false. Otherwise, if the initial state is false, then the pointer is at `1`, where LZ77 finds a match `1!0` in the dictionary and moves its pointer just after the gadget, changing the state to true.

- Alternatively, one could use lazy matching to construct a NOT gadget; here we only sketch the general idea in our example setting. A possible NOT gadget after `secret=1` is `ABCDEF`, where we add `1ABC`, `BCDE`, and `DEF` to the dictionary. If the initial state is true, then LZ77 first matches `ABC`, and then replaces it with a lazy match `BCDE`, reaching the false state at `F`; otherwise, if the initial state is false, then LZ77 subsequently matches `1ABC` and `DEF`, reaching the true state.

  This method creates more nuisance than the previous method, but may help reduce the chance of failure if we cannot use a custom character for NOT gadgets.

6.4.2 AND-match

An AND-match gadget sets the state as the logical AND of the initial state and the result of a match gadget. More specifically, if the initial state is true, then the AND-match gadget sets the state according to the result of the match gadget; otherwise, if the initial state is false, then the AND-match gadget sets the state to false.

For simplicity, we only consider AND-match gadgets constructed from byte-match gadgets in this section.

AND byte-match

The AND-match gadget constructed from a byte-match gadget $P\|g$ is simply $S = P\|g$. We could nonetheless use append a dummy gadget (Section 6.2.2) to the previous gadget if some of the requirements below are not satisfied, or the concatenation of the AND-match gadget with the previous gadget may create unwanted interference. Let $\ell_s = |S|$. We add $S[0, \ell_s - 1] = g[0] || P$ to a dictionary.

In our example setting, if we want to perform the AND operation on two byte-match gadgets `secret=1` and `data=2`, by treating the latter as an AND-match gadget, we could simply write `secret=1data=2` and put `1data=` into a dictionary. If we want to separate them, we could add a dummy gadget like `{R22}` to make it `secret=1{R22}data=2`, for which we should instead put `1{R22`, `{R22}`, and `}data=` into the dictionaries.

For our construction to work correctly, we require that LZ77 cannot find a match for $S[0, \ell_s]$ in the sliding window.

If the initial state is true, then the LZ77 pointer is at $S[1]$ (in our example, `d` in `data=`), so $S[1, \ell_s]$ acts as a byte-match gadget; otherwise, the initial state is false, then the pointer is at $S[0]$ (`1` in `secret=1`), and LZ77 finds the match for $S[0, \ell_s - 1]$ (`1data=`) we added to a dictionary, moving the pointer to $S[\ell_s]$ (`2` in `data=2`), which is the state of false.

This construction should be used with care, because it adds a non-random entry to a dictionary.

6.4.3 OR-match

An OR-match gadget sets the state as the logical OR of the initial state and the result of a match gadget. More specifically, if the initial state is true, then the OR-match gadget sets the state to true; otherwise, if the initial state is false, the OR-match gadget sets the state according to the result of the match gadget.

For simplicity, we only consider OR-match gadgets constructed from byte-match gadgets.

OR byte-match

Unlike AND byte-match, we do need to prepend a string before the byte-match gadget to perform the OR operation. We could still append a dummy gadget to the previous gadget to provide separation.

Here, we describe one possible way of building an OR byte-match gadget when lazy matching is enabled. We note that a similar but simpler construction exists when lazy matching is disabled.

Let $P||g$ be a byte-match gadget for the prefix P and guess g, where $\ell_p \geq \text{min\_match} + 2$ for $\ell_p := |P|$. An OR byte-match gadget of length ℓ_s constructed from the byte-match gadget can be $S = T||P||g$, where T is a string of length ℓ_t such that $\ell_t \geq \mathsf{min\_match}$, and, if using zlib, $\ell_t \leq \mathsf{max\_lazy}$. The string T can be drawn at random, preferably containing a special character reserved for OR-match; ideally, at least $T[2]$ should be a special character. Let k be a positive integer such that $2 \leq k \leq \ell_p - \mathsf{min\_match}$. We add three strings to the dictionaries:

$$\begin{aligned} S[0, \ell_t] &= S[0] \| T \\ S[2, \ell_t + k] &= T[2, \ell_t] \| P[1, k] \\ S[\ell_t + k + 1, \ell_s] &= P[k + 1, \ell_p] \| g \end{aligned}$$

For the underlying byte-match gadget to work correctly, we may also add P to a dictionary if LZ77 cannot find P in the sliding window.

In our example setting, to perform the OR operation on `secret=1` and `secret=2`, we could choose $\ell_t = 3$ and $k = 5$, and let `e|Asecret=2` be the OR byte-match gadget. The concatenation is then `secret=1e|Asecret=2`, and we should add strings `1e|A`, `|Asecre`, and `t=2` to arbitrary dictionaries.

For this gadget to work correctly, we should ensure that LZ77 does not find a match for $S[1, \ell_t + 1]$, $S[3, \ell_t + k + 1]$ or $S[\ell_t + k + 2, \ell_s + 1]$ in the sliding window.

If the initial state for the OR-match gadget is true, then the LZ77 pointer is at $S[1]$, where LZ77 finds a match for $S[1, \ell_t]$ in a dictionary (likely in the entry for $S[0, \ell_t]$). Because $\ell_t \leq \mathsf{max\_lazy}$ in zlib, LZ77 performs lazy matching and finds $S[2, \ell_t + k]$ of length $\ell_t + k - 1 > \ell_t$. Because this lazy match is longer and cannot be further improved either by extension or lazy matching, LZ77 takes the lazy match and advances the pointer to $S[\ell_t + k + 1]$. There, LZ77 finds $S[\ell_t + k + 1, \ell_s]$ in a dictionary, which meets the minimum match length and cannot be further improved, making the LZ77 pointer advance just past the gadget to the state of true.

Otherwise, if the initial state for the OR-match gadget is false, then the LZ77 pointer is at $S[0]$, where LZ77 finds the match $S[0, \ell_t]$ in a dictionary, which cannot be further improved. Therefore, the pointer moves to $S[\ell_t + 1]$, the start of the byte-match gadget. Assuming that the strings we added do not interfere with the byte-match gadget, it should set the state depending on whether LZ77 can find $P \| g$ in the sliding window.

In our example, starting at `e` in `e|A`, LZ77 first finds a match for `e|A`, then replaces it with the lazy match `|Asecre`, and finally matches `t=2`, moving the pointer just after `secret=2`; starting at `1` in `secret=1`, LZ77 matches `1e|A`, and the pointer falls at `s` in `secret=2`.

This construction should be used with great care, because it adds two non-random entries to the dictionaries.

6.5 Example programs

In this section, we describe three types of programs for compression side-channel attacks.

6.5.1 CRIME chain

An attacker can use a CRIME chain to check with high confidence whether a character follows a prefix in the plaintext, or check whether a string exists in the plaintext. We have already used CRIME chains implicitly in Chapter 5.

The construction of a CRIME chain is straightforward. Its gadgets are of the form

$$(\mathsf{align}\|)\mathsf{match}\|\mathsf{amplify}$$

where align, match and $\mathsf{amplify}$ are respectively an optional align gadget, a match gadget and an amplify gadget.

6.5.2 CRIME slide

A CRIME slide allows an attacker to perform a parallel search on multiple candidate characters that follow a prefix in the plaintext, or perform a parallel search on multiple target strings. The CRIME slide can be seen as an extension to the divide-and-conquer technique (Section 3.5.2) for compression side-channel attacks. Its gadgets are of the form

$$(\mathsf{align}\|)\mathsf{match\text{-}1}\|\mathsf{or\text{-}match\text{-}2}\|\mathsf{or\text{-}match\text{-}3}\|\ldots\|\mathsf{or\text{-}match\text{-}}n\|\mathsf{amplify}$$

where align is an optional align gadget, $\mathsf{match\text{-}1}$ is a byte-match gadget for the first candidate, $\mathsf{or\text{-}match\text{-}}i$ is an OR-match gadget constructed from the byte-match gadget for the i-th candidate for each $i \in \{2, \ldots, n\}$, and $\mathsf{amplify}$ is an amplify gadget.

If at least one candidate can be found in the sliding window, then the initial state for $\mathsf{amplify}$ is true; otherwise, the initial state for $\mathsf{amplify}$ is false, creating a difference in their compressed lengths.

If OR-match gadgets are unusable for some reason, an attacker could perform a parallel search by composition. The composition can be done at either the level of gadgets or the level or programs. However, these constructions require longer queries and are likely unstable.

An attacker can construct the gadgets as

$$(\mathsf{align}\|)\mathsf{match\text{-}1}\|\mathsf{amplify\text{-}1}\|\mathsf{align}\|\mathsf{match\text{-}2}\|\mathsf{amplify\text{-}2}\|\ldots\|\mathsf{align}\|\mathsf{match\text{-}}n\|\mathsf{amplify\text{-}}n$$

where for each $i \in [n]$, $\mathsf{match}\text{-}i$ is a byte-match gadget for the i-th candidate, and $\mathsf{amplify}\text{-}i$ is an amplify gadget. If we assume every amplify gadget increases the compressed length by around the same value at the state true, then an attacker could deduce from the compressed length how many candidates are in the plaintext. Furthermore, if each amplify gadget changes the compressed length differently, then an attacker may learn more information from the compressed length, and may even be able to deduce which candidate is in the plaintext by making a single query.

Alternatively, an attacker could concatenate multiple CRIME chains by making the query

$$Q = \mathsf{crime\text{-}chain\text{-}1} \| \mathsf{crime\text{-}chain\text{-}2} \| \ldots \| \mathsf{crime\text{-}chain}\text{-}n$$

where for each $i \in [n]$, $\mathsf{crime\text{-}chain}\text{-}i$ refers to the CRIME chain constructed for the i-th candidate. This construction achieves a similar effect as the construction above.

6.5.3 CRIME cascade

A CRIME cascade can not only match multiple candidates in parallel, but also inform the attacker the first matched candidate from the length of the compressed plaintext. Its gadgets are of the form

$$(\mathsf{align}\|)\mathsf{match\text{-}1}\|\mathsf{amplify\text{-}1}\|\mathsf{or\text{-}match\text{-}2}\|\mathsf{amplify\text{-}2}\|\ldots\|\mathsf{or\text{-}match}\text{-}n\|\mathsf{amplify}\text{-}n.$$

where align is an optional align gadget, match-1 is a byte-match gadget for the first candidate, for each $i \in \{2, \ldots, n\}$, $\mathsf{or\text{-}match}\text{-}i$ is an OR-match gadget constructed from the byte-match gadget for the i-th candidate, and for each $i \in [n]$, $\mathsf{amplify}\text{-}i$ is an amplify gadget.

Assume that each amplify gadget increases the compressed length by around the same value d at the state true. Then, for each $i \in [n]$, if the i-th candidate is the candidate with the smallest index that appears in the sliding window, then for each $j \in [n]$, $\mathsf{amplify}\text{-}j$ increases the compressed length by around d if and only if $j \geq i$, meaning that the compressed length is increased by around $(n - i + 1) \cdot d$. Therefore, an attacker could learn from the compressed length how many amplify gadgets produced an increased length, and then naturally deduce the first matched candidate.

We note that the alternative constructions for the CRIME slide can achieve the same effect by using amplification gadgets of different lengths, but the resulting queries would be longer than a CRIME cascade.

Chapter 7

Discussion and future work

7.1 Discussion

It is hard to gauge the real-world implications of our work due to its theoretical nature. While we are currently unaware of any deployed system that is simultaneously not vulnerable to existing compression side-channel attacks and can be exploited using our new techniques, we reasonably expect that such targets would soon be found if their vendors do not take proper measures to mitigate the potential compression side channels in time. Therefore, we urge vendors not to perform compression before encryption in their products whenever possible.

While previous works have suggested that adding random noise can be effective at mitigating compression side-channel attacks, our amplification techniques imply that these mitigations are not as effective as claimed. However, the exact extents to which they are effective are still unknown. Similarly, systems that are not highly interactive by nature may also be more vulnerable to compression side-channel attacks than previously expected.

Finally, this thesis concurs with [29] that the compression side channel is different in nature from many other side channels. We therefore advocate for a change of mindset when analysing compression side-channel attacks. In particular, a slightly more fitting name for the attacks could be compression (length) oracle attacks.

7.2 Future work

Some possible directions for future work include:

- *Evaluation.* Due to time constraints, this thesis did not include an evaluation of our techniques. Therefore, a primary task for future work is to conduct a rigorous evaluation, including but not limited to the effects of different parameters, compression levels and compressor implementations.

- *Techniques.* Some of our techniques can likely be further refined, and it should be possible to find new techniques with a better understanding of the Huffman coding process in DEFLATE/zlib.
- *Attacks.* It remains an open problem to demonstrate the practicality of our new techniques on a real-world target, which likely requires additional tweaks to adapt to the attack scenario.
- *Extensions.* It is of particular interest to extend our results to other compressors. Some variants of our amplification techniques already apply to several other compressors, like brotli and bzip2. However, one may uncover more interesting techniques by inspecting their algorithms and implementations more closely. In addition, our techniques can clearly be adapted to exploit the timing side channels in DEFLATE/zlib decompressors [50].
- *Bounds.* Apart from extending our results on randomised padding, it is also of interest to study the effect of random noise in compressors and masking secrets, which may require further extensions to our models.

Appendix A

Generic lower bounds on sensitivity

We give some lower bounds on the sensitivity of generic compressors.

Definition A.1 (Neighbors) *Let Σ be an alphabet, $\ell \in \mathbb{N}_+$, $S \subseteq \Sigma^\ell$. The neighbors of the set S are*

$$N[S] := \{a \in \Sigma^\ell : \exists b \in S \wedge \mathsf{HD}(a,b) \leq 1\},$$

and for every $k \in \mathbb{N}$, the k-neighbors of the set S are

$$N^{(k)}[S] := \{a \in \Sigma^\ell : \exists b \in S \wedge \mathsf{HD}(a,b) \leq k\}.$$

Definition A.2 (Hamming ball) *Let Σ be an alphabet, $\ell \in \mathbb{N}_+$, $a \in \Sigma^\ell$, $k \in \{0,1,\dots,\ell\}$. The Hamming ball on Σ^ℓ with the center a and radius k is*

$$\mathsf{Ball}(a,k) := \{b \in \Sigma^\ell : \mathsf{HD}(a,b) \leq k\}.$$

Theorem A.3 *Let C be a compressor on an input alphabet Σ of size σ and an output alphabet Ω of size $\omega > 1$. Let $\alpha \in (0,1)$, $\beta \in (0, 1-\alpha)$, and $\ell \in \mathbb{N}_+$ such that $\ell \geq \omega^{1/(1-\alpha-\beta)}$. If there exists $s \in \Sigma^\ell$ such that $\Pr[|C(s)| \leq \ell^\alpha - 1] \neq 0$, then*

$$\Delta(C,\ell) > \frac{\beta \log \ell + \log(\sigma - 1)}{\log \omega}.$$

Proof If C is randomised, then there exist random coins $r \in \mathcal{R}_{C,\ell}$ such that $|C(s;r)| \leq \ell^\alpha - 1$, and we consider the compressor C', obtained by fixing the random coins to r for inputs of length ℓ; that is,

$$C'(m) := \begin{cases} C(m;r) & \text{if } m \in \Sigma^\ell, \\ C(m) & \text{otherwise.} \end{cases}$$

Clearly, C' is deterministic on inputs in Σ^ℓ, and $\Delta(C,\ell) \geq \Delta(C',\ell)$.

We assume in the rest of the proof that C is deterministic on inputs in Σ^ℓ.

Suppose, for contradiction, that $\Delta(C,\ell) \le (\beta \log \ell + \log(\sigma-1)) / \log \omega$.

Let $k \in \mathbb{N}$, with its exact value to be specified later. Consider the Hamming ball on Σ^ℓ with the center s and radius k. We have

$$|\mathsf{Ball}(s,k)| = \sum_{i=0}^{k} \binom{\ell}{i} (\sigma-1)^i > \binom{\ell}{k} (\sigma-1)^k > \left(\frac{(\sigma-1)\ell}{k}\right)^k. \tag{A.1}$$

Because $|C(s)| \in \mathbb{N}$, $|C(s)| \le \lfloor n^\alpha \rfloor - 1$. By Lemma 4.7, $\forall x \in \mathsf{Ball}(s,k)$,

$$|C(x)| \le |C(s)| + k \cdot \Delta(C,\ell) \le \lfloor \ell^\alpha \rfloor - 1 + k \cdot \Delta(C,\ell).$$

Let $h = \lfloor \ell^\alpha \rfloor - 1 + k \cdot \Delta(C,\ell)$. Since $\omega \ge 2$ and C is injective on Σ^l,

$$|\mathsf{Ball}(s,k)| \le \left| \bigcup_{i=1}^{h} \Omega^i \right| = \sum_{i=1}^{h} \omega^i < \omega^{h+1} \le \omega^{\lfloor \ell^\alpha \rfloor + k\cdot\Delta(C,\ell)}. \tag{A.2}$$

Let $k = \lfloor \ell^\alpha \rfloor$. We have

$$\begin{aligned}
\log|\mathsf{Ball}(s,k)| &> k(\log \ell + \log(\sigma-1) - \log k) && (\textit{Eq.}\ (\text{A.1}))\\
&= k((1-\beta)\log \ell - \log k) + k(\beta \log \ell + \log(\sigma-1)) &&\\
&\ge k((1-\beta)\log \ell - \log k) + k \log \omega \cdot \Delta(C,\ell) && (\text{supposition})\\
&\ge \lfloor \ell^\alpha \rfloor ((1-\beta)\log \ell - \log\lfloor \ell^\alpha \rfloor) + k \log \omega \cdot \Delta(C,\ell) && (k = \lfloor \ell^\alpha \rfloor)\\
&\ge \lfloor \ell^\alpha \rfloor (1-\alpha-\beta)\log \ell + k \log \omega \cdot \Delta(C,\ell) && (\alpha > 0)\\
&\ge \lfloor \ell^\alpha \rfloor \log \omega + k \log \omega \cdot \Delta(C,\ell) && (\ell \ge \omega^{1/(1-\alpha-\beta)})\\
&> \log|\mathsf{Ball}(s,k)|. && (\textit{Eq.}\ (\text{A.2}))
\end{aligned}$$

We arrived at a contradiction. □

Remark A.4 *Asymptotically better results are likely unattainable with our proof technique. As a matter of fact, the sensitivity of the Kolmogorov complexity (i.e. the shortest program that outputs a given string) is $O(\log \ell)$ [1], so an asymptotically better bound for generic compressors may be out of reach.*

Corollary A.5 *Let C be a compressor on input alphabet $\{0,1\}$ and output alphabet $\{0,1\}$. Let $\alpha \in (0,1)$, $\beta \in (0, 1-\alpha)$, and $\ell \in \mathbb{N}_+$ such that $\ell \ge 2^{1/(1-\alpha-\beta)}$. If there exists $s \in \Sigma^\ell$ such that $\Pr[|C(s)| \le \ell^\alpha - 1] \ne 0$, then $\Delta(C,\ell) > \beta \log \ell$.*

We can extend Corollary A.5 by relating it to the vertex-isoperimetric problem on a hypercube.

Lemma A.6 (corollary of Harper's theorem [20]) *Let $\ell \in \mathbb{N}_+$, $S \subseteq \{0,1\}^\ell$. If there exists $k \in [\ell - 1]$ such that*

$$|S| \ge \sum_{i=0}^{k} \binom{\ell}{i},$$

then

$$|N[S]| \ge \sum_{i=0}^{k+1} \binom{\ell}{i}.$$

Theorem A.7 *Let C be a deterministic compressor on input alphabet $\{0,1\}$ and output alphabet $\{0,1\}$. Let $\alpha \in (0,1)$, $\beta \in (0, 1-\alpha)$, $\ell \in \mathbb{N}_+$ such that $\ell \geq 2^{1/(1-\alpha-\beta)}$, and $h \in [\lfloor \ell^\alpha \rfloor]$. If there exists $S \subseteq \{0,1\}^\ell$ such that $|S| \geq \sum_{i=0}^{h} \binom{\ell}{i}$, and $\forall s \in S$, $|C(s)| \leq \lfloor \ell^\alpha \rfloor + h\beta \log \ell - 1$, then $\Delta(C, \ell) > \beta \log \ell$.*

Proof (sketch) The basic idea is to show that having such a set S is equivalent in effect to having a single string s such that $|C(s)| \leq \ell^\alpha - 1$. Then, we can follow the steps in Theorem A.3 to complete the proof.

Suppose, for contradiction, that $\Delta(C, \ell) \leq \beta \log \ell$. Let $k = \lfloor \ell^\alpha \rfloor$.

By repeatedly applying Lemma A.6 for $m = k - h$ times, we get

$$\left| N^{(m)}[S] \right| \geq \sum_{i=0}^{h+m} \binom{\ell}{i} = \sum_{i=0}^{k} \binom{\ell}{i} \geq \left(\frac{\ell}{k}\right)^k .$$

On the other hand, for every $x \in N^{(m)}[S]$, there exists $s \in S$ such that $\mathsf{HD}(s, x) \leq m$. By Lemma 4.7,

$$\begin{aligned} |C(x)| &\leq |C(s)| + m \cdot \Delta(C, \ell) \\ &\leq \lfloor \ell^\alpha \rfloor + h\beta \log \ell - 1 + (k-h) \cdot \Delta(C, \ell) \\ &\leq \lfloor \ell^\alpha \rfloor + k\beta \log \ell - 1. \end{aligned}$$

Since C is injective,

$$\left| N^{(m)}[S] \right| \leq 2^{\lfloor \ell^\alpha \rfloor + k\beta \log \ell}.$$

Applying a similar reasoning as Theorem A.3 shows that $\Delta(C, \ell) > \beta \log \ell$.$\square$

Corollary A.8 *Let C be a compressor on input alphabet $\{0,1\}$ and output alphabet $\{0,1\}$. Let $\alpha \in (0,1)$, $\beta \in (0, 1-\alpha)$, $\gamma \in (0,1]$, $\ell \in \mathbb{N}_+$ such that $\ell \geq 2^{1/(1-\alpha-\beta)}$, and $h \in [\lfloor \ell^\alpha \rfloor]$. If there exists $S \subseteq \{0,1\}^\ell$ such that $|S| \geq \sum_{i=0}^{h} \binom{\ell}{i} / \gamma$, and $\Pr[|C(s)| \leq \lfloor \ell^\alpha \rfloor + h\beta \log \ell - 1 \mid s \hookleftarrow\$ S] \geq \gamma$, then $\Delta(C, \ell) > \beta \log \ell$.*

Proof (sketch) For all random coins $r \in \mathcal{R}_{C,\ell}$, let

$$S_r = \{s \in S : |C(s; r)| \leq \lfloor \ell^\alpha \rfloor + h\beta \log \ell - 1\}.$$

Suppose, for contradiction, that $\forall r \in \mathcal{R}_{C,\ell}$, we have $|S_r| < \gamma |S|$. Then

$$\begin{aligned} \Pr[|C(s)| \leq \lfloor \ell^\alpha \rfloor + h\beta \log \ell - 1 \mid s \hookleftarrow\$ S] &= \sum_{s \in S} \frac{1}{|S|} \Pr[s \in S_r \mid r \hookleftarrow\$ \mathcal{R}_{C,\ell}] \\ &= \sum_{r \in \mathcal{R}_{C,\ell}} \frac{1}{|\mathcal{R}_{C,\ell}|} \frac{|S_r|}{|S|} < \gamma, \end{aligned}$$

and we arrived at a contradiction. Therefore, there exists $r \in \mathcal{R}_{C,\ell}$ such that $|S_r| \geq \gamma |S| \geq \sum_{i=0}^{h} \binom{\ell}{i}$. Similar to the proof for Theorem A.3, we complete our proof by first fixing the random coins used by C to r for inputs of length ℓ, which does not increase $\Delta(C, \ell)$, and then applying Theorem A.7. $\square$

Appendix B

Additional proofs

B.1 Proof for Theorem 4.3

Proof For all $q \in \mathbb{N}$ such that $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \neq 0$, by definition, there exists a LOCI adversary $\mathcal{A}$ that makes at most q queries, such that $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L}(\mathcal{A}) > 0$.

Without loss of generality, assume $\Pr[\mathsf{LOCI}(\mathcal{A}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] \geq 1/2$. If $\Pr[\mathsf{LOCI}(\mathcal{A}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] = 1/2$, then $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L}(\mathcal{A})$ would be 0, so we have $\Pr[\mathsf{LOCI}(\mathcal{A}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] > 1/2$.

Let $\mathcal{R}_{\mathcal{A}}$ be the set of all random coin tosses for $\mathcal{A}$, and for $r \in \mathcal{R}_{\mathcal{A}}$, let $\mathcal{A}_r$ denote running $\mathcal{A}$ with fixed random coins r; i.e. $\mathcal{A}_r() = \mathcal{A}(;r)$. We have

$$\begin{aligned}\Pr[\mathsf{LOCI}(\mathcal{A}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] &= \sum_{r \in \mathcal{R}_{\mathcal{A}}} \Pr[r] \Pr[\mathsf{LOCI}(\mathcal{A}_r, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] \\ &\leq \max_{r \in \mathcal{R}_{\mathcal{A}}} \Pr[\mathsf{LOCI}(\mathcal{A}_r, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}].\end{aligned}$$

Therefore, there exists $r' \in \mathcal{R}_{\mathcal{A}}$, such that

$$\Pr[\mathsf{LOCI}(\mathcal{A}_{r'}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] \geq \Pr[\mathsf{LOCI}(\mathcal{A}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] > \frac{1}{2}. \quad \text{(B.1)}$$

Note that for $x \in \{0, 1\}$, the game LOCI for adversary $\mathcal{A}_{r'}$ conditioned on $b = x$ can be simulated perfectly by a deterministic algorithm. Therefore,

$$\Pr[\mathsf{LOCI}(\mathcal{A}_{r'}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true} \mid b = x] \in \{0, 1\},$$

and thus

$$\Pr[\mathsf{LOCI}(\mathcal{A}_{r'}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] \in \left\{0, \frac{1}{2}, 1\right\}.$$

By Eq. (B.1), we have $\Pr[\mathsf{LOCI}(\mathcal{A}_{r'}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] = 1$. Because $\mathcal{A}_{r'}$, as an instantiation of $\mathcal{A}$, also makes at most q queries,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} &\geq \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L}(\mathcal{A}_{r'}) \\ &= |2\Pr[\mathsf{LOCI}(\mathcal{A}_{r'}, C, ck_0, ck_1, L) \Rightarrow \mathsf{true}] - 1| \\ &= 1.\end{aligned}$$

We complete the proof by noting that $\mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q} \leq 1$ by definition. □

B.2 Proof for Theorem 4.35

Proof It is easy to see $\mathsf{Adv}^{\mathsf{LOCI}}_{C_{\mathsf{pad}},ck_0,ck_1,L,q} \leq \mathsf{Adv}^{\mathsf{LOCI}}_{C,ck_0,ck_1,L,q}$ via a straightforward reduction from a LOCI adversary against C_{pad} to a LOCI adversary against C. We show in the remaining of the proof that $\mathsf{Adv}^{\mathsf{LOCI}}_{C_{\mathsf{pad}},ck_0,ck_1,L,q} \leq \mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta_{k,L},q}$.

Let $\mathcal{A}$ be a LOCI adversary against C_{pad}, ck_0, ck_1 and L that makes at most q queries. Without loss of generality, we assume that $\mathcal{A}$ does not make invalid queries, i.e. for every query $O(m', m'')$ that $\mathcal{A}$ makes, $m', m'' \in \Sigma^*$, and $|m'| + |m''| \leq L - k$.

We construct from $\mathcal{A}$ a BD-HIDE adversary $\mathcal{B}$ against $\mathcal{D}_{\mathsf{pad}}$ and $\delta_{k,L}$ as in Fig. B.1.

```
Adversary $\mathcal{B}^{\mathsf{Hide}}()$
1 :  $b' \leftarrow\$ \mathcal{A}^{O}(ck_0, ck_1)$
2 :  return $b'$

Oracle $O(m', m'')$
1 :  $m_0 \leftarrow m' \| ck_0 \| m''$
2 :  $m_1 \leftarrow m' \| ck_1 \| m''$
3 :  $r \leftarrow\$ \mathcal{R}_{C,\ell}$
4 :  $z_0 \leftarrow C(m_0; r)$
5 :  $z_1 \leftarrow C(m_1; r)$
6 :  $d \leftarrow |z_1| - |z_0|$
7 :  $h \leftarrow\$ \mathsf{Hide}(d)$
8 :  return $|z_0| + h$
```

Figure B.1: The BD-HIDE adversary $\mathcal{B}$ constructed from $\mathcal{A}$.

By definition, the BD-HIDE adversary $\mathcal{B}$ makes at most q queries, and

$$|d| = ||C(m_0; r)| - |C(m_1; r)|| \leq \Delta_{\mathsf{HD}(ck_0,ck_1)}(C, |m_0|) \leq \delta_{k,L},$$

so the adversary $\mathcal{B}$'s answer to the oracle query $O(m', m'')$ made by $\mathcal{A}$ is

$$|z_0| + \ell = |z_0| + b \cdot d + p = |z_0| + b \cdot (|z_1| - |z_0|) + p = |z_b| + p,$$

where $z_b \leftarrow\!\!\$\; C(m' \| ck_b \| m'')$, and p is sampled independently for each query from $\mathcal{D}_{\mathsf{pad}}$. Therefore, the $\mathsf{BD\text{-}HIDE}$ adversary $\mathcal{B}$ simulates perfectly the LOCI game for $\mathcal{A}$, and thus

$$\mathsf{Adv}^{\mathsf{LOCI}}_{C_{\mathsf{pad}},ck_0,ck_1,L}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta}(\mathcal{B}) \leq \mathsf{Adv}^{\mathsf{BD\text{-}HIDE}}_{\mathcal{D}_{\mathsf{pad}},\delta,q}. \qquad \square$$

Bibliography

[1] Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *Inf. Comput.*, 291:104999, 2023.

[2] Janaka Alawatugoda, Douglas Stebila, and Colin Boyd. Protecting encrypted cookies from compression side-channel attacks. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 86–106. Springer, Heidelberg, January 2015.

[3] Martin R. Albrecht, Raphael Eikenberg, and Kenneth G. Paterson. Breaking bridgefy, again: Adopting libsignal is not enough. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 269–286. USENIX Association, August 2022.

[4] Tal Be'ery and Amichai Shulman. A perfect CRIME? only TIME will tell. *Black Hat Europe*, 2013.

[5] Colin Boyd. Enhancing secrecy by data compression: Theoretical and practical aspects. In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 266–280. Springer, Heidelberg, April 1991.

[6] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. *SRS Research Report*, 124, 1994.

[7] Yu Long Chen, Wonseok Choi, and Changmin Lee. Improved multi-user security using the squared-ratio method. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 694–724. Springer, Heidelberg, August 2023.

[8] Wei Dai, Viet Tung Hoang, and Stefano Tessaro. Information-theoretic indistinguishability via the chi-squared method. In Jonathan Katz and

Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 497–523. Springer, Heidelberg, August 2017.

[9] Jean Paul Degabriele. Hiding the lengths of encrypted messages via gaussian padding. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1549–1565. ACM Press, November 2021.

[10] L. Peter Deutsch. DEFLATE compressed data format specification version 1.3. RFC 1951, 1996.

[11] L. Peter Deutsch. GZIP file format specification version 4.3. RFC 1952, 1996.

[12] L. Peter Deutsch and Jean loup Gailly. ZLIB compressed data format specification version 3.3. RFC 1950, 1996.

[13] Andrés Fábrega, Carolina Ortega Pérez, Armin Namavari, Ben Nassi, Rachit Agarwal, and Thomas Ristenpart. Injection attacks against end-to-end encrypted applications. *IEEE Symposium on Security and Privacy*, 2024. To appear.

[14] Antaeus Feldspar. An explanation of the *Deflate* algorithm. https://web.archive.org/web/20240324020205/https://www.zlib.net/feldspar.html.

[15] Nils Fleischhacker, Kasper Green Larsen, and Mark Simkin. How to compress encrypted data. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 551–577. Springer, Heidelberg, April 2023.

[16] 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*, pages 655–672. USENIX Association, August 2016.

[17] Kai Gellert, Tibor Jager, Lin Lyu, and Tom Neuschulten. On fingerprinting attacks and length-hiding encryption. Cryptology ePrint Archive, Report 2021/1027, 2021. https://eprint.iacr.org/2021/1027.

[18] Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. Bit catastrophes for the burrows-wheeler transform. In *DLT*, volume 13911 of *Lecture Notes in Computer Science*, pages 86–99. Springer, 2023.

[19] Yoel Gluck, Neal Harris, and Angelo Prado. BREACH: Reviving the CRIME attack. *Black Hat*, 2013.

[20] Lawrence H. Harper. Optimal numberings and isoperimetric problems on graphs. *Journal of Combinatorial Theory*, 1(3):385–393, 1966.

[21] Mathew Hogan, Yan Michalevsky, and Saba Eskandarian. DBREACH: Stealing from databases using compression side channels. In *2023 IEEE Symposium on Security and Privacy*, pages 182–198. IEEE Computer Society Press, May 2023.

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

[23] Kahil Jallad, Jonathan Katz, and Bruce Schneier. Implementation of chosen-ciphertext attacks against PGP and GnuPG. In *ISC*, volume 2433 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2002.

[24] Mark Johnson, David Wagner, and Kannan Ramchandran. On compressing encrypted data without the encryption key. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 491–504. Springer, Heidelberg, February 2004.

[25] Dimitris Karakostas, Aggelos Kiayias, Eva Sarafianou, and Dionysis Zindros. CTX: Eliminating BREACH with context hiding. *Black Hat EU*, 2016.

[26] Dimitris Karakostas and Dionysis Zindros. Practical new developments on breach. *Black Hat Asia*, 2016.

[27] Jonathan Katz and Bruce Schneier. A chosen ciphertext attack against several E-mail encryption protocols. In Steven M. Bellovin and Greg Rose, editors, *USENIX Security 2000*. USENIX Association, August 2000.

[28] James Kelley and Roberto Tamassia. Secure compression: Theory & practice. Cryptology ePrint Archive, Report 2014/113, 2014. https://eprint.iacr.org/2014/113.

[29] John Kelsey. Compression and information leakage of plaintext. In Joan Daemen and Vincent Rijmen, editors, *FSE 2002*, volume 2365 of *LNCS*, pages 263–276. Springer, Heidelberg, February 2002.

[30] Demijan Klinc, Carmit Hazay, Ashish Jagmohan, Hugo Krawczyk, and Tal Rabin. On compression of data encrypted with block ciphers. Cryptology ePrint Archive, Report 2010/477, 2010. https://eprint.iacr.org/2010/477.

[31] Tadayoshi Kohno. Attacking and repairing the winZip encryption scheme. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick McDaniel, editors, *ACM CCS 2004*, pages 72–81. ACM Press, October 2004.

[32] Boris Köpf and David A. Basin. An information-theoretic model for adaptive side-channel attacks. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 286–296. ACM Press, October 2007.

[33] Charles 'Buck' Krasic, Mike Bishop, and Alan Frindell. QPACK: Field compression for HTTP/3. RFC 9204, 2022.

[34] Guillaume Lagarde and Sylvain Perifel. Lempel-Ziv: a "one-bit catastrophe" but not a tragedy. In *SODA*, pages 1478–1495. SIAM, 2018.

[35] James I. Lathrop and Martin Strauss. A universal upper bound on the performance of the Lempel-Ziv algorithm on maliciously-constructed data. In *SEQUENCES*, pages 123–135. IEEE, 1997.

[36] Blake Loring. A solution to compression oracles on the web. `https://web.archive.org/web/20240228135557/https://blog.cloudflare.com/a-solution-to-compression-oracles-on-the-web/`.

[37] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, 1996.

[38] Rafael Palacios, Andrea Fariña Fernández-Portillo, Eugenio Fco. Sánchez-Úbeda, and Pablo García-De-Zúñiga. HTB: A very effective method to protect web servers against BREACH attack to HTTPS. *IEEE Access*, 10:40381–40390, 2022.

[39] Jacques Patarin. The "coefficients H" technique. In *Selected Areas in Cryptography*, volume 5381 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 2008.

[40] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389. Springer, Heidelberg, December 2011.

[41] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from Threema: Analysis of a secure messenger. In *USENIX Security Symposium*, pages 1289–1306. USENIX Association, 2023.

[42] Brandon Paulsen, Chungha Sung, Peter A. H. Peterson, and Chao Wang. Debreach: Mitigating compression side channels via static analysis and transformation. In *ASE*, pages 899–911. IEEE, 2019.

[43] Roberto Peon and Herve Ruellan. HPACK: Header compression for HTTP/2. RFC 7541, 2015.

[44] Alfredo Pironti, Pierre-Yves Strub, and Karthikeyan Bhargavan. Identifying website users by tls traffic analysis: New attacks and effective countermeasures. Technical Report RR-8067, 2012.

[45] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking S/MIME and OpenPGP email encryption using exfiltration channels. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 549–566. USENIX Association, August 2018.

[46] Angelo Prado, Neal Harris, and Yoel Gluck. SSL, GONE IN 30 SECONDS. https://web.archive.org/web/20240215060338/https://breachattack.com/.

[47] Ivan Ristic. Defending against the BREACH attack. https://web.archive.org/web/20231130135655/https://blog.qualys.com/product-tech/2013/08/07/defending-against-the-breach-attack.

[48] Juliano Rizzo and Thai Duong. The CRIME attack. *Ekoparty*, 2012.

[49] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, New York, second edition, 1996.

[50] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical timing side-channel attacks on memory compression. In *2023 IEEE Symposium on Security and Privacy*, pages 1186–1203. IEEE Computer Society Press, May 2023.

[51] Claude E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

[52] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*, pages 27–42. Springer, 2010.

[53] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[54] Nick Sullivan. CRIME, TIME, BREACH and HEIST: A brief history of compression oracle attacks on HTTPS. https://web.archive.org/web/20231207195011/https://www.helpnetsecurity.com/2016/08/11/compression-oracle-attacks-https/.

[55] Cihangir Tezcan and Serge Vaudenay. On hiding a plaintext length by preencryption. In Javier Lopez and Gene Tsudik, editors, *ACNS 11*, volume 6715 of *LNCS*, pages 345–358. Springer, Heidelberg, June 2011.

[56] Mathy Vanhoef and Tom Van Goethem. HEIST: HTTP Encrypted Information can be stolen through TCP-windows. *Black Hat USA*, 2016.

[57] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *2008 IEEE Symposium on Security and Privacy*, pages 35–49. IEEE Computer Society Press, May 2008.

[58] Charles V. Wright, Lucas Ballard, Fabian Monrose, and Gerald M. Masson. Language identification of encrypted VoIP traffic: Alejandra y roberto or alice and bob? In Niels Provos, editor, *USENIX Security 2007*. USENIX Association, August 2007.

[59] Michał Zieliński. SafeDeflate: compression without leaking secrets. Cryptology ePrint Archive, Report 2016/958, 2016. https://eprint.iacr.org/2016/958.

[60] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.