Criminology: Refined Techniques for Compression Side-Channel Attacks*

Yuanming Song, Lenka Mareková, and Kenneth G. Paterson

Applied Cryptography Group, ETH Zurich
{yuanming.song,lenka.marekova,kenny.paterson}@inf.ethz.ch

September 10, 2026

Abstract. It has been known for two decades that performing compression before encryption is dangerous, because it introduces a side channel leaking information about plaintexts through ciphertext lengths: the compressed plaintext length may be visible in the ciphertext length, and the amount of compression obtained is plaintext-dependent; hence an adversary can obtain some leakage about the plaintext via observation of ciphertext lengths. This issue was first pointed out by Kelsey (FSE 2002) and turned into a practical plaintext recovery attack in the form of the CRIME attack on SSL and TLS by Rizzo and Duong in 2012. A long series of variations and attacks against other systems followed. Despite the known dangers, the compress-then-encrypt paradigm is still prevalent in practice today. This may be because the compression-based side channel is susceptible to noise and may require a large number of queries to enable plaintext recovery, and so can be mitigated by either adding noise (e.g. with random padding) or limiting an adversary's interaction with the system.

We demonstrate that this side channel is much more powerful than previously thought. We focus on the widely-used DEFLATE algorithm in our analysis. We present novel techniques that enable strong amplification of small length differences arising during compression. Our *telescoping* and *chaining* amplification techniques exploit the way in which DEFLATE replaces common strings by shorter back-references. Our *collision-based amplification* technique focusses on exploiting hash table collisions in DEFLATE implementations. This involves a deeper examination (and exploitation) of the internals of DEFLATE than in previous works. These insights result in compressed length differences growing *linearly* with the length of queries. Compared with length differences of a few bits or bytes in prior work, our new amplification techniques thus enable us to defeat existing noise-based countermeasures.

Finally, we introduce the concept of *CRIME automata*, these being carefully crafted query strings that enable an attacker to exert fine control over the internal behaviour of DEFLATE and produce differences in the output lengths of the compressor according to various criteria (such as whether the DEFLATE sliding window contains a given target string). In turn, our automata are composed in a modular fashion from *gadgets* having different functions, including matching against target strings, performing logical operations between other gadgets, and, most importantly, amplifying differences in output lengths using the above-mentioned techniques. We provide multiple, concrete automata designs that serve different attack goals. These designs are supported by experiments and a publicly available codebase demonstrating the power, flexibility, and practical impact of our CRIME automata approach.

* This is the full version of a work to appear at ACM CCS 2026.

Table of Contents

1 Introduction . . . 4
1.1 Contributions . . . 5
1.2 Ethical considerations . . . 6
1.3 Roadmap . . . 6
2 Related work . . . 6
2.1 Compression and encryption . . . 6
2.2 Compression side-channel attacks . . . 7
2.3 Mitigations . . . 9
2.4 Security models . . . 11
2.5 Robustness of compressors . . . 12
2.6 Weird machines . . . 12
3 Background . . . 13
3.1 Terminology . . . 13
3.2 Notation . . . 13
3.3 DEFLATE . . . 14
3.3.1 LZ77 . . . 14
3.3.2 Huffman coding . . . 16
3.4 zlib . . . 17
3.5 DEFLATE implementation zoo . . . 19
4 Existing techniques . . . 23
4.1 Kelsey's secret extraction attacks . . . 23
4.2 CRIME/BREACH . . . 24
5 Attack model and assumptions . . . 26
5.1 Attack model . . . 26
5.2 Assumptions . . . 27
5.3 Running example . . . 28
6 General amplification . . . 28
6.1 Overview of amplification . . . 28
6.2 Telescoping . . . 29
6.3 Chaining . . . 32
6.3.1 Distance-based chaining . . . 33
6.3.2 Match-based chaining . . . 36
7 Collision-based amplification . . . 42
7.1 Collisions . . . 42
7.2 Step-by-step introduction . . . 45
7.2.1 Basic collision-based chaining . . . 46
7.2.2 Extending the chain . . . 48
7.2.3 A more efficient extension . . . 49
7.2.4 Basic bootstrapping . . . 50
7.2.5 Advanced bootstrapping . . . 51
7.2.6 Automating the technique . . . 52
7.3 Templates for deflate_slow . . . 53
7.3.1 deflate_slow instance set . . . 53
7.3.2 Main template . . . 55
7.3.3 Small diverse double-colliding set . . . 61
7.3.4 No diverse double-colliding set . . . 62
7.3.5 No double-colliding set . . . 62
7.3.6 Alternative instance set constraints . . . 63
7.4 Templates for deflate_fast . . . 66
7.4.1 deflate_fast instance set . . . 66
7.4.2 Adapting templates to deflate_fast . . . 66
7.4.3 max_chain = 4 . . . 68
7.4.4 max_chain = 1 . . . 68
8 CRIME automata . . . 69

8.1 Overview . . . 69
8.2 Separation gadgets . . . 71
8.2.1 Align gadget . . . 71
8.2.2 NOP gadget . . . 73
8.3 Match gadgets . . . 74
8.3.1 Match gadget . . . 74
8.3.2 Unmatch gadget . . . 75
8.3.3 Telescoped-match gadget . . . 76
8.4 Amplify gadgets . . . 76
8.5 Logical gadgets . . . 76
8.5.1 NOT gadget . . . 76
8.5.2 AND-match . . . 79
8.5.3 OR-match . . . 80
8.6 Example automata . . . 83
9 Experimental evaluation . . . 85
9.1 Implementation . . . 85
9.2 Experimental setup . . . 87
9.3 Results . . . 88
9.3.1 Amplification . . . 88
9.3.2 Secret recovery . . . 90
10 Discussion . . . 91

1 Introduction

Compression and encryption are commonly performed on data in transit and data at rest. Compression saves bandwidth or storage, and encryption protects data confidentiality. However, it is 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* [Kel02]. Namely, an attacker can typically deduce the approximate length of the compressed plaintext from the ciphertext length, which intuitively reflects the redundancy of the plaintext. While seemingly innocuous, the compression side channel enables an attacker with the ability to adaptively embed queries in the plaintext to recover secrets in the plaintext. Prominent examples of compression side-channel attacks include CRIME [RD12] and BREACH [GHP13], which can recover secrets such as cookies by exploiting compression in TLS and HTTP respectively.

We sketch the core idea of a compression side-channel attack described by Kelsey [Kel02] against the DEFLATE compression algorithm [Deu96a], which forms the basis of most compression side-channel attacks. DEFLATE is used in popular compressors such as zlib [RGA] and Gzip [Fre]. It works by replacing repeated substrings in the plaintext with back-references to their previous occurrences. Suppose the attacker can adaptively choose a query Q to embed in the plaintext together with $P\|ck$, the concatenation of a known prefix P and a fixed secret ck that is the target of the attack, and learn the length of the compressed plaintext. To recover the first byte of the secret, the attacker makes queries of the form $Q = P\|g$, the concatenation of the prefix and a guess g for the first byte of the secret ck. The query with the correct guess corresponds to a plaintext with a one-byte longer repeated substring and likely results in a slightly shorter compressed length. The attacker makes a query for each possible guess, and the unknown byte is likely the guess that yields the shortest ciphertext. By repeating this process byte-by-byte, the attacker can eventually recover the entire secret. The attack as described involves small differences in the observed ciphertext lengths and requires many queries to reliably recover each plaintext byte.

Given that compression side-channel attacks have been known for more than twenty years and include high-profile examples like CRIME and BREACH, the pervasiveness of compression before encryption may come as a surprise. Consequently, researchers continue to develop compression side-channel attacks against a variety of real-world systems, including databases [HME23, BME25], encrypted messaging applications [PST23, FPN+24] and password managers [FNA+24]. More concerningly, even after vulnerability disclosures, the affected vendors often do not implement any mitigation at all [HME23, FPN+24] or only partially mitigate the compression side channel by introducing noise [FNA+24].

A probable reason for vendors to continue performing compression before encryption is that compression side channels, like other side channels, are often perceived to be "noisy" and hard to exploit in practice. Indeed, compression side-channel attacks typically require hundreds or thousands of queries to extract a short secret, and many more if any noise is present, making these attacks costly and prone to detection. At the same time, disabling compression would usually incur drastic overhead in storage or communication. Therefore, it is tempting for vendors to regard compression side channels as theoretical concerns with little or no real-world impact.

We believe that the compression side channel is not yet well understood. The research community's current knowledge of the compression side channel is closely tied to concrete attacks like CRIME and BREACH, which, as proof of concepts, only needed to explore the details of the 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"* [GHP13, Section 4.1]. Many attacks also rely on heuristics for which the designers themselves can at best offer informal explanations [RD12, GHP13, BS13].

With this work, we challenge an implicit assumption in previous works on compression side-channel attacks: the attacker must rely on tiny differences in compressed lengths to learn information about the secret. Indeed, in the example sketched earlier, the compressed length likely differs by no more than a few bytes between correct and incorrect guesses, which means the attack is highly susceptible to noise. This assumption is sometimes made explicit by justifying the addition of noise as an effective defence [Deg21, PFSG22]. It likely stems from the intuition that similar plaintexts have similar redundancies and therefore should compress to similar lengths by the same compression algorithm. This, however, is typically not a design goal of compressors and in fact does not hold for many compressors [AFI23]: it is known that small input differences can lead to large differences in the compressed output lengths.

It could be tempting to solve the problem of compression side channels by simply introducing more noise. The amount could be based on the above-mentioned theoretical upper bounds on the sensitivity

of the compressor in question, i.e. the maximum difference in compressed lengths with regard to small changes in the plaintext. However, existing works on compressor sensitivity (e.g. [AFI23,BLG25,FSNI25]) exclusively study theoretical compression algorithms, while real-world compressor implementations can be much more complex. In Sections 6 and 7, we design and analyse practical techniques to amplify compressed length differences. Section 6 exhibits amplification techniques exploiting the way in which DEFLATE replaces common strings by shorter back-references. On the other hand, Section 7 focusses on exploiting hash table collisions in DEFLATE implementations. This involves a deeper examination (and exploitation) of the internals of DEFLATE than in previous works on compression-based attacks. Our results in Section 7, in particular, exceed the theoretical upper bound on the sensitivity of LZ77 under much weaker conditions, and the compressed length difference grows about linearly with the query length, to hundreds of megabytes or more if there is no limit to the query length. These results enable us to obtain query-efficient, plaintext-recovering compression side-channel attacks that overcome existing noise-based countermeasures.

One may further question whether the compression side channel should be considered as a side channel at all. 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. Moreover, while the operation of the compression algorithm may be complex, it is deterministic in nature and its output length is exactly measurable.

An implication of the above comparison is that the compression side channel may be studied and controlled to a very high level of precision. As an illustration of this precision, we will show in Section 8 that we can extract a deterministic state machine (with a single bit state) from the DEFLATE algorithm. We can manipulate the state of this machine through carefully crafted input strings that we call *gadgets*; these effectively apply logical operators to the state according to properties of the gadgets. For example, we can set the state according to whether a certain substring of the gadget matches a target, unknown string in the input in our so-called *match gadget*, and we can use an *amplify gadget* to magnify the difference in compressed lengths depending on the value of the state. As we will also show, these gadgets can be chained together to allow complex logical operations to be performed on the state, with related, controlled impacts on the compressed output length. We dub this approach *CRIME automata*. It allows us to modularly design efficient queries with advanced functionalities that were not anticipated in previous works. For example, we can build queries that test for multiple possible candidate byte values in a single attempt, while simultaneously amplifying the compressed length difference to overcome any practical noise-based countermeasure.

Our automata designs are supported by experiments that validate their correctness and demonstrate that they lead to improved plaintext recovery attacks, even in the face of significant levels of noise. For example, in the presence of uniform-length noise of up to 100B, our CRIME slide automaton can recover a 32-byte alphanumeric secret cookie with 214.9 queries on average. Furthermore, we can reduce the average number of queries to 119.9 by using CRIME cascade automata, even outperforming binary search without any noise in responses.

We emphasise that we introduce the CRIME automata concept as a means to more clearly express our results; validation of our approach comes from careful analysis of DEFLATE and experiments rather than an application of the theory of automata. Determining the computational power of CRIME automata could be an interesting theoretical question that we defer to future work.

1.1 Contributions

We are the first to rigorously study compression side channels in DEFLATE compressors, and our work improves the state of the art for compression side-channel attacks in several aspects. Our main contributions are as follows:

1. We design and analyse novel techniques to reduce or remove the effect of noise in compression side-channel attacks against DEFLATE and its implementations by amplifying differences in compressed lengths with specially crafted queries.
2. We introduce the novel paradigm of CRIME automata. Here, we exploit DEFLATE as an abstract state machine, using this insight in combination with gadgets to modularly design queries that allow us to precisely control the information leakage arising from the compression side channel. In particular, our techniques allow an attacker to successfully recover target plaintext bytes while consuming fewer queries than the best known previous attacks and while overcoming noise.

3. We describe an implementation framework for performing compression side-channel attacks using our techniques. The framework leverages our automata abstraction to define queries in a modular way, and allows automating the selection of gadget parameters, which lowers the barrier to evaluating and applying our techniques.

In addition, our amplification techniques give the first examples on the lack of robustness for DEFLATE, i.e. we show that the length of the compressed data may change significantly given only small changes in the input. Our surveys of compression side-channel attacks and defences in Section 2 and DEFLATE implementations in Section 3 are the most comprehensive to our knowledge and may be of independent interest.

1.2 Ethical considerations

Compression side-channel attacks like CRIME and BREACH are widely known in the security community, and performing compression before encryption is generally recognised as bad practice. As a result, we are currently unaware of any compression side channels in deployed systems that are unknown to their vendors. By publishing this paper, we aim to change the vendors' perception of compression side channels and prompt them to mitigate existing compression side channels in their systems.

Following the examples of previous works on compression side-channel attacks [GHP13, KZ16, Naf18, HME23, BME25], we choose to make our framework for mounting compression side-channel attacks publicly available, in order to help other researchers understand, evaluate, apply and improve our techniques. Unlike previous works, our framework assumes an abstract target and cannot be directly used to attack any deployed system. We believe that the benefit outweighs the potential harm of making our techniques more accessible to malicious actors, who could in any case reproduce the attacks solely from our paper.

Since our techniques could potentially be used to improve the existing compression side-channel attacks against WhatsApp and KeePassXC [FPN+24, FNA+24], we shared a draft of our paper with WhatsApp and KeePassXC on August 4, 2026. Both vendors have acknowledged receipt.

1.3 Roadmap

We first review the related work in Section 2 and provide the background knowledge for our technical results in Section 3. We introduce some existing techniques for compression side-channel attacks in Section 4 and present our attack model in Section 5.

We then describe and analyse our refined attack techniques on DEFLATE compressors, including general amplification (Section 6), collision-based amplification (Section 7) and CRIME automata (Section 8). We introduce our implementation framework and evaluate our techniques in Section 9. Finally, we discuss the implications of our results and conclude in Section 10.

2 Related work

In this section, we group the related work into several categories and review them separately. A paper may contribute to more than one category (e.g. [FBNR25, BLG25, GHP13]), in which case we discuss the paper multiple times from different angles.

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 [Sha49], 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 [Boy91, MvOV96, Sch96]. Another argument is that compression can help resist cryptanalysis. For example, Katz and Schneier [KS00] 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 [JKS02]. Recently, Truong et al. [TMS+25] reported that compression was the main obstacle to their fingerprinting attacks against content-defined chunking schemes.

However, Kelsey [Kel02] 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.

Implementations of compression algorithms are seldom, if at all, constant-time; as a result, they could be vulnerable to timing or cache side-channel attacks. Schwarzl et al. [SBS+23] demonstrated several attacks that recover encrypted data via information leakage from *decompression* time differences. Recently, Minkin and Kasikci [MK24] discovered cache side channels in zlib [RGA], ncompress [Fry] and bzip2 [Sew] compressor implementations by applying a taint-tracking-based tool they designed, and they demonstrated two cache side-channel attacks on bzip2.

Apart from side-channel attacks, there are more examples illustrating the danger of performing compression before encryption. Kohno [Koh04] gave a chosen-ciphertext attack against the WinZip[1] encryption scheme by modifying the unauthenticated compression method field. Garman et al. [GGK+16] described a gzip format oracle attack against Apple iMessage. Poddebniak et al. [PDM+18] showed that while compression significantly complicates their EFAIL attack against OpenPGP, it also allows an attacker to more precisely modify and exfiltrate the plaintext. Albrecht et al. [ABJM21] described a padding oracle attack against Bridgefy, a mesh messaging application, exploiting the composition of RSA PKCS#1 v1.5 encryption and Gzip compression.

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. [GJLN22] suggested that performing compression before encryption can help mitigate fingerprinting attacks, and remarked that "the general recommendation to disable compression before encryption in all applications seems overgeneralized" [GJLN22, Section 5].

Finally, we note that there is work on the counter-intuitive idea of compressing encrypted data without using the secret key [FLS23, JWR04, KHJ+10] by applying homomorphic encryption or distributed source coding. Further discussion is out of the scope of this paper.

2.2 Compression side-channel attacks

Kelsey [Kel02] 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.

The "CRIME family". *CRIME*, developed by Rizzo and Duong [RD12], 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 Langley was the first to sketch a possible compression side-channel attack against SPDY and HTTPS to extract secret cookies,[2] before the CRIME attack was announced.

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* [BS13] 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* [VVG16] 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* [GHP13] attack by Gluck et al. also targeted compressed HTTPS responses as TIME did, but it required attackers to directly observe the sizes of HTTP responses. Gluck et al. described several techniques to reduce noise and improve stability, and discussed some possible mitigations to the attack.

Karakostas and Zindros [KZ16] built *Rupture*, a framework for performing compression side-channel attacks. They employed statistical methods to reduce noise and bypass block alignment, and proposed several optimisations, including divide-and-conquer and browser parallelization. We remark that Gluck et al. have suggested that statistical methods can be used to mitigate noise [GHP13, Section 3.1], and

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

Pornin was likely the first to introduce the divide-and-conquer method for improving the compression side-channel attack.[3]

Note that the attacks above all target compressors implementing the DEFLATE algorithm [Deu96a], which we will introduce in Section 3.3. All of these attacks exploit the attacker's ability to co-locate chosen plaintext close to the target plaintext in the string that is compressed before being encrypted; our work is in the same setting. Readers interested in the relations between these attacks can find more details in [Sul16].

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

Gupta [Gup17] was the first to identify that compression in VPN protocols may introduce compression side channels. Nafeez [Naf18] demonstrated the VORACLE attack, a compression side-channel attack that exploits LZ77-based compression on HTTP website traffic in OpenVPN.

Hogan et al. [HME23] 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 used repeated queries to amplify the noisy signal in order to extract secrets from the database. In addition to DEFLATE, DBREACH also works on LZ4 [LZ4] and Snappy [Sna], two compressors based on the LZ77 algorithm [ZL77]. Recently, Bourassa et al. [BME25] improved the performance of DBREACH attacks by speeding up compressibility score computation with binary search and testing multiple guesses at the same time, and extended the targets of DBREACH attacks to data marked as deleted in the database.

Paterson et al. [PST23] 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. [FPN+24] 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. In another paper, Fábrega et al. [FNA+24] described a compression side-channel attack against the KeePassXC password manager that recovers URLs and usernames from a set of known strings, which they stated to be significantly more practical than the aforementioned attack in [FPN+24]. The attack exploited zlib compression in KeePassXC, which compresses database files to the gzip format. KeePassXC mitigated the attack by including a freshly generated random array in the database when the database is saved, which Fábrega et al. considered to be a partial mitigation as it could be bypassed using statistical techniques.

Automated analysis. We briefly discuss previous works on performing automated analysis of compression side channels using symbolic execution and fuzzing, which have largely been overlooked by later studies on compression side-channel attacks. While technically interesting, the current results fail to match the performance of manual analysis and are costly in computation.

Bang et al. [BAP+16] studied a type of side channels with "segmented oracles", described the best adversary strategy, and applied symbolic execution to analyse two side channels, including a noiseless compression side channel in a custom implementation of LZ77 with about 60 lines of Java code (see [PBP+17, Fig. 7]). However, Bang et al. did not provide any concrete argument for the optimality of their best adversary; in fact, it is clearly suboptimal for the compression side channel they targeted.[4]

Phan et al. [PBP+17] used an automated approach to find the optimal attack and its greedy approximations against the same compression side channel in [BAP+16] and found the same attack as in [BAP+16], because they computed the attacks that maximise the number of observations instead of more relevant measures like Shannon entropy (which [PBP+17] claimed to be computationally infeasible in practice) or min-entropy, and only computed the optimal attacks for a few steps due to heavy computational costs. Nevertheless, with unlimited computational resources, it should be possible to discover an optimal attack subject to certain conditions using the technique described in [PBP+17]. A related

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

[4] As an example, Bang et al. showed in an experiment that their best adversary can recover a secret of length 3 with alphabet size 4 after 10 queries [BAP+16, Section 6.4], whereas a divide-and-conquer attacker (which is also suboptimal in general; see Section 8) can already recover the same secret after 6 queries.

work by Bang et al. [BRB18] presented a symbolic execution technique to perform side-channel attacks in noisy environments, which could potentially be applied to compression side channels with noise.

Nilizadeh et al. [NNP19] presented a fuzzer for automatically discovering side channels using differential fuzzing. Generally speaking, the fuzzer tries to find a public value and two secret values that maximise the difference in some specified metric when the program is executed on the same public value and different secret values. Nilizadeh et al. used the tool to discover a compression side channel in a compressor implementation similar to that in [BAP+16] and reported a maximum output length difference of 782 bytes found by the fuzzer; however, the evaluation apparently did not set non-trivial limits to the size of secrets.[5] Therefore, the large length difference does not imply new techniques for compression side-channel attacks, as it can be achieved or even surpassed with two very long secrets, one compressible and another incompressible. This is consistent with the note by Nilizadeh et al. that the side channels discovered by the fuzzer may not be exploitable in practice [NNP19, Section III.E].

Traffic analysis. There is also a line of research on leveraging compression side channels for traffic analysis of protocols, such as TLS [PSB12] and encrypted VoIP [WBC+08, WBMM07]. More recently, Albrecht et al. [AEP22] 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 paper, 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, whereas a traffic analysis attacker typically does not have control over the plaintext and is therefore more limited from a technical perspective.

Hardware compression side channels. Besides in software, compression is also used in hardware to improve performance, which can also lead to compression side channels. Tsai et al. [TSFS20] analysed the security of cache compression and developed a compression side-channel attack that can efficiently recover secret data in cache. Vicarte et al. [VSN+21] analysed the security of some proposed microarchitectural optimisations, including pipeline compression and register-file compression. Wang et al. [WPG+24] investigated graphical data compression in integrated GPUs and developed a cross-origin pixel stealing attack exploiting GPU compression.

Hardware compression side channels are out of the scope of this paper, because compression algorithms in hardware are typically customised as opposed to general-purpose like DEFLATE.

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. [Ris13] for web servers.

Separating secrets. Separating secrets from user inputs or not compressing secrets can mitigate or potentially eliminate the compression side channel [ASB15, GHP13, KH25, KKSZ16, KS12, Lor18, PSPW19]. However, this approach either requires developers to manually mark or move secret fields [KKSZ16], which may be impractical [GHP13], or requires a mechanism to automatically identify secrets [ASB15, KH25, Lor18, PSPW19], which can be error-prone and might introduce additional side channels if implemented badly. In addition, compressing secrets separately may still leak information, especially if different users control different secrets located in the same plaintext [KH25].

[5] See https://github.com/isstac/diffuzz/blob/v1.0.0-citable/evaluation/stac_crime_unsafe/src/CRIME_Driver.java and https://github.com/isstac/diffuzz/blob/v1.0.0-citable/evaluation/stac_crime_unsafe/02_startFuzzer.sh.

Compartmentalization. Several recent works suggested partitioning the plaintext into different buckets and compressing each bucket separately [BME25, FNA+24, FPN+24, HME23], which we refer to as compartmentalization. For example, compression side-channel attacks against a database can be mitigated by compressing each row of the database separately, as attackers would then need to control elements from the same row as the secret, which is much harder in many cases.

For compartmentalization, the choice of buckets is crucial for both security and performance, which requires care from developers. Fábrega et al. [FBNR25] proposed using view-based partitioning to mitigate compression side-channel attacks, which, simply speaking, works by partitioning the plaintext into parts that are visible to different users, so that the input of a malicious user is likely separated from the secrets of other users. They developed a framework for view-based partitioning, formally analysed its security, and demonstrated its efficiency on WhatsApp and KeePassXC. However, this approach does not apply in the original setting of the CRIME attack.

We note that compartmentalization typically does not eliminate compression side channels, because a passive attacker may still learn information from the lengths of the compressed buckets. In addition, if the attacker can co-locate queries with the secret in the same bucket, for example by exploiting incorrect partitions or bugs, then they will get access to a "clean" compression side channel with less interference from irrelevant data. Finally, for unstudied systems, it is unclear whether compartmentalization can mitigate the compression side channel without incurring significant storage or communication overhead.

Switching compressors. Most existing compression side-channel attacks target the nearly ubiquitous LZ77-based compressors. This suggests that switching to a different compressor might help. However, as Kelsey [Kel02] remarked, compressors using adaptive Huffman or Markov coding can be attacked using similar techniques, and "the same side channels clearly exist and can be exploited" [Kel02, Section 8] in compressors utilising the Burrows-Wheeler transform [BW94].

Compressors that only adapt to the input to a small extent are likely less vulnerable to compression side-channel attacks. Kihara and Shimizu [KS12] suggested that using a static dictionary for compression can be an effective mitigation of the CRIME attack and its variants. Alawatugoda et al. [ASB15] described a compressor with a fixed dictionary and provided a security analysis. However, this compressor performs relatively poorly, and the analysis is flawed (see [Son24, Section 4.5.1]). Zieliński [Zie16] 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.

One may also modify the compression algorithm to introduce non-determinism to the compression process, but a careful analysis is necessary no matter how complex the modifications appear. Yang and Gong [YG19] proposed randomising the LZ77 algorithm as follows: the compression algorithm first compresses the plaintext using LZ77, then randomly duplicates some symbols in the LZ77 output, and finally decompresses the modified output and compresses again using LZ77. However, Yang and Gong did nots specify the corresponding decompression algorithm, and it appears to us that no perfectly correct decompression algorithm exist, even with knowledge of the random coins used in compression.

HPACK [PR15] and QPACK [KBF22] 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 the CRIME attack, using fixed Huffman coding and restricted dynamic dictionaries. Note that HPACK and QPACK are not general-purpose compressors, but tailored for HTTP header compression.

Masking secrets. Gluck et al. [GHP13] 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. Levillain et al. [LGD15] proposed two countermeasures against attacks on TLS based on masking, one of which is similar to the aforementioned mitigation in [GHP13], and claimed that these countermeasures are effective against several attacks, including compression side-channel attacks, but their claimed effectiveness against RC4 biases was later falsified in [PS18]. Django,[6] a Python-based web framework, masks CSRF tokens as a defence against BREACH attacks [Djaa].

Despite being easy to implement and intuitively secure, masking has several drawbacks. First, developers have to identify secrets in advance; second, masking secrets increases the length of the compressed data, especially if secrets are non-random or many secrets are present [GHP13, KZ16]; finally, although

[6] https://www.djangoproject.com/

intuitively secure, masking secrets has not received a formal analysis in the context of compression side-channel attacks.

Adding noise. The introduction of noise to the input, the internal states, or the output of a compressor was proposed to make the compression side channel harder to exploit [Kel02,GHP13,KS12]. In this paper, we do not classify mitigations that modify the internals of a compressor as adding noise, but rather as switching compressors. 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 [GHP13, KS12, KZ16, YG19].

Degabriele [Deg21] studied several random padding schemes in the context of length-hiding encryption. Degabriele showed that Gaussian padding is generally more effective than uniform padding when multiple samples are available, and analysed the effects of different padding schemes on a variant of the CRIME attack.

Palacios et al. [PFSG22] proposed HTB, a mitigation against the BREACH attack using random-length padding. HTB works by modifying the Gzip library to add a random-length fake filename (which does not get compressed) on each compression. Palacios et al. showed with a simplified model and experiments that HTB significantly increases the number of queries an attacker needs to perform. HTB is recommended on the website of the BREACH attack [PHG13].

Concurrent to our work, Blocki et al. [BLG25] noted that the amount of random padding added in [PFSG22] may not be large enough. They introduced the notion of differentially private compression and proposed a general construction based on the pad-then-encrypt composition.

Limiting interaction. Rate limiting and monitoring can slow down the BREACH attack [GHP13]; 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 [GHP13], performing referrer checks [Ris13] and disabling third-party cookies [KZ16].

Combining compression with encryption. Kelsey suggested preprocessing the input by applying a stream cipher that "generates a keystream of extremely low Hamming weight" [Kel02], which, besides being a weak form of masking, might be regarded as an encrypt-then-compress-then-encrypt construction. Yang and Gong [YG19] implemented and evaluated this construction.

It might be tempting to consider combining compression with encryption in non-trivial ways to achieve better security. However, as Kelley and Tamassia [KT14] observed, a (perfectly correct) cryptosystem cannot be both be IND-CPA secure and achieve meaningful general-purpose compression. Note that the squeeze cipher designed by Kelley and Tamassia [KT14], 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.

A recent work by Gross et al. [GKO+25] proposed a scheme for integrated compression and encryption by repeatedly introducing small changes in Huffman or arithmetic coding based on the secret key. Gross et al. claimed that their scheme is resistant to compression side-channel attacks like BREACH, but their description of BREACH and argument suggest a misunderstanding of compression side-channel attacks. The paper does not perform any standard security analysis or report the compression ratios achieved by their scheme.

2.4 Security models

There are numerous works on formally modelling side-channel attacks. For example, Köpf and Basin [KB07] 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 [KT14] 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. [ASB15] commented, the work of Kelley and Tamassia does not capture compression side-channel attacks.

Alawatugoda et al. [ASB15] defined several new security notions for compression side-channel attacks targeting 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 proved that separating secrets from user inputs achieves CCI security, and analysed the CR security of their fixed-dictionary fixed-width compression scheme.

Fábrega et al. [FBNR25] developed a new security model for injection attacks, which also includes a large subset of compression side-channel attacks. They proved that injection capabilities do not give an attacker much additional advantage against their view-based partitioning framework. To this end, they first showed that the injection attack security of a scheme can be reduced to the security of the underlying encryption algorithm and the length independence of the scheme, and then showed that their framework achieves perfect length independence.

Security models for length-hiding encryption [Deg21, GJLN22, PRS11, TV11] 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. [GJLN22] 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 [Deg21] revisited the previous work [PRS11, TV11] 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.

2.5 Robustness of compressors

Our amplification techniques in Sections 6 and 7 are related to studies on the robustness of compressors. Informally speaking, a compressor is robust if the length of the compressed data never changes significantly over small changes in its input. Intuitively, if the target compressor is robust, then the compressed length is almost constant for the same query with different secrets, which means the attacker cannot significantly amplify the compression side channel. However, the reverse direction is not necessarily true, as a compressor that is not robust in general may still exhibit robustness if the input has a certain format, and counterexamples to robustness do not necessarily lead to compression side-channel attacks.

The term "one-bit catastrophe" refers to the phenomenon that changing a single bit or character in the input of some compressors incurs a drastic change in the length of the compressed data. Motivated by earlier works such as [LS97], Lagarde and Perifel [LP18] studied the one-bit catastrophe for the LZ78 algorithm, and Giuliani et al. [GIL+23] studied this phenomenon for the Burrows-Wheeler transform.

Akagi et al. [AFI23] 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 additive and multiplicative sensitivities of various theoretical compressors. As we remark in Section 6.2, our telescoping technique is similar to the technique used in [AFI23] to derive lower bounds on the sensitivities of several LZ-family compressors. Blocki et al. [BLG25] improved the bounds by Akagi et al. on the additive sensitivity of the LZ77 algorithm as in [ZL77], and Fujie et al. [FSNI25] obtained asymptotically tight bounds on the additive sensitivities of several LZ-family compressors.

Previous works on the robustness of compressors mostly studied theoretical compression algorithms,[7] whereas real-world compressor implementations like zlib typically chain several compression algorithms together and heavily modify them for better performance. As we will see in Section 7, the sensitivity of LZ77 in most DEFLATE implementations significantly exceeds the theoretical upper bound on the sensitivity of LZ77 in [ZL77].

2.6 Weird machines

Our concept of CRIME automata in Section 8 is inspired by weird machines in security literature. Generally speaking, a weird machine is some input crafted by the attacker that results in certain computations being performed on the target machine by inducing directed state changes [BLP+11].

[7] The only exception we are aware of is an example shown by Inenaga on the high sensitivity of the 7-Zip compression software; see Footnote 62, Section 6.2.

A famous example is Return-Oriented Programming (ROP) [Sha07], which allows the attacker to bypass data execution protection using crafted gadgets that point to short sequences of instructions in libc. Some recent works have constructed weird machines for microarchitectural side-channel attacks [EBE+21, HRY24, Kap23, KKC+23, WBP+26, WPWB24]. Our work can be viewed as identifying weird machines in the DEFLATE algorithm and using them to mount compression side-channel attacks.

3 Background

3.1 Terminology

We list some important terminology that we use in this paper.

- *Compression:* In this paper, compression refers specifically to lossless data compression.
- *Compression side channel:* The leakage of information about the plaintext from the length of the compressed data. Our work does not cover cache or timing side channels in compression algorithms [MK24, SBS+23].
- *DEFLATE*: We refer to DEFLATE as a general compression algorithm that conforms to the algorithm details in the DEFLATE compressed data format specification [Deu96a, Section 4].
- *LZ77*: When not otherwise specified, we refer to LZ77 as a variant of the LZ77 algorithm [ZL77] described in the DEFLATE specification [Deu96a, Section 4]. Note that LZ77 behaves differently for different compression strategies used in DEFLATE; for simplicity, we may refer to LZ77 with some particular compression strategy directly as LZ77 if it is clear from the context.
- *zlib*: A library that implements DEFLATE compressors and decompressors [RGA]. Note its difference to the zlib compressed data format [DlG96].
- *GNU Gzip* (or *Gzip* in short): A software tool for data compression using DEFLATE [Fre]. Note its difference to the gzip compressed data format [Deu96b].

We note that compression algorithms, compressor implementations and compressed data formats are used interchangeably in many sources, which may be inaccurate in the context of compression side-channel attacks.

3.2 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 $n \in \mathbb{N}$, let $[n] = \{1, 2, \ldots, n\}$. We use $\log x$ to denote the binary logarithm of x for $x > 0$.

For $x, y \in \mathbb{N}$, let $x \oplus y$ denote the bitwise-XOR of x and y, which we define recursively as follows:

$$x \oplus y = \begin{cases} ((x+y) \bmod 2) + 2\left(\left\lfloor \frac{x}{2} \right\rfloor \oplus \left\lfloor \frac{y}{2} \right\rfloor\right) & \text{if } x > 1 \text{ or } y > 1, \\ (x+y) \bmod 2 & \text{otherwise.} \end{cases}$$

An *alphabet* is a finite non-empty set Σ, where the size of the alphabet is $|\Sigma|$. A *string* S on Σ is an an element of the set Σ^*, comprising a sequence of *characters* in Σ. Let ε denote the empty string. Let $\Sigma^+ = \Sigma^*\backslash\{\varepsilon\}$. Let $|S|$ denote the *length* of a string S, and the length of the empty string is 0. For simplicity, we sometimes represent a one-character string with the character it contains.

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 $i, j \in [\ell]$ with $i \le j$, let $S[i:j] = S[i]S[i+1]\ldots S[j]$. For strings S and T, let $S\|T$ denote their concatenation. We say a string S is a *substring* of another string T, if there exists an integer i with $1 \le i \le |T| - |S| + 1$, such that $S = T[i : i + |S| - 1]$. For a string S and an integer $n \in \mathbb{N}$, let $S^n = \underbrace{S\|S\|\cdots\|S}_{n \text{ times}}$.

A *byte* is an integer in $\{0, 1, \ldots, 255\}$; a *byte string* is a string in which every character is a byte. For a byte string S, let $\mathsf{uintl}(S)$ be the integer representation of S in little-endian, and let $\mathsf{uintb}(S)$ be the integer representation of S in big-endian; that is, $\mathsf{uintl}(S) = \sum_{i=1}^{|S|} S[i] \cdot 2^{8(i-1)}$, and $\mathsf{uintb}(S) = \sum_{i=1}^{|S|} S[i] \cdot 2^{8(|S|-i)}$.

A byte b is *ASCII-printable* if $b \in \{32, \ldots, 126\}$; a string is ASCII-printable if every character of the string is an ASCII-printable byte. For simplicity, we may represent ASCII-printable bytes with their corresponding ASCII characters.

We write `<Lx,Dy>` for a *back-reference* of length x and distance y in the output of the LZ77 algorithm, where we may replace y with $*$ for an unspecified distance.[8]

[8] Note that different occurrences of $*$ in the LZ77 output generally represent different distances.

3.3 DEFLATE

We focus on compressors that conform to the algorithm details described in the specification of the widely-used DEFLATE format [Deu96a, Section 4], such as zlib and GNU Gzip. Therefore, we refer to DEFLATE as a general compression algorithm rather than a compressed data format. The exact details of the DEFLATE algorithm may vary for different implementations and parameter choices (see Section 3.5), and readers can refer to our description of zlib in Section 3.4 for a concrete DEFLATE implementation.

Note that, for historical as well as for performance reasons, there exist compressor implementations that are not based on the DEFLATE algorithm but can still produce compressed data in the DEFLATE format, such as Zopfli [Zop]. These compressors are out of the scope of this paper.

Very roughly speaking, the DEFLATE algorithm works as follows. First, it runs a variant of the LZ77 algorithm [ZL77] to scan the input data for repeated strings, for each repetition creating a back-reference to its last occurrence. Then, it uses Huffman coding [Huf52] to further compress the LZ77 output.

Our description of DEFLATE does not cover its full details; we refer interested readers to the DEFLATE specification [Deu96a] and an explanation of DEFLATE by Feldspar [Fel].

3.3.1 LZ77

LZ77 [ZL77] is a compression algorithm that works by replacing repeated strings in the input data with back-references. Many variants of the LZ77 algorithm exist, and in this paper, unless otherwise specified, we refer to LZ77 as the variant described in the DEFLATE specification [Deu96a, Section 4].[9]

The LZ77 algorithm operates in a streaming manner on byte strings. It scans the input data with a pointer for repetitions, keeping a sliding window of $\mathsf{window_size}$ 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 it prefers the one that is the nearest to the pointer. LZ77 ignores any match that is shorter than the minimum length of $\mathsf{min_match} = 3$ bytes or longer than the maximum length of $\mathsf{max_match} = 258$ bytes. If LZ77 finds a match in the sliding window, then it 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
```

gives us

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

where we write `<L`x`,D`y`>` 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, which we will cover next, and we assume the algorithm always searches the whole sliding window for the longest match.

Hash table. To help find repetitions, the LZ77 algorithm maintains a chained hash table containing the positions in the sliding window. Each position represents a 3-byte substring of the input that starts at this position, whose hash value is used to store the position in the hash table. Substring positions on each hash chain are ordered by their proximity to the LZ77 pointer. The oldest entries may be purged when adding new positions to the hash table. The DEFLATE specification [Deu96a] does not specify how to add a new position to the hash table; however, it can be inferred that LZ77 inserts a substring position before the head of its corresponding chain in the hash table and sets it as the new head.

To find the longest match, LZ77 computes the hash value of the 3-byte substring starting from the pointer, and traverses the chain in the hash table that corresponds to the hash value. For each entry on the chain that stores a position representing 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, with the output being the longest match found, if any.

[9] Some may find this LZ77 variant closer to the LZSS algorithm [SS82].

Algorithm 1: A simplified version of the LZ77 algorithm.

Input: A byte string S.
Output: A sequence of bytes and back-references $(t_1, t_2, \ldots, t_m)$.
$p \leftarrow 1$;
$m \leftarrow 0$;
while $p \leq |S|$ **do**
  $m \leftarrow m + 1$;
  $(l, d) \leftarrow (0, 0)$;
  for $j \leftarrow 1$ **to** window_size **do**
    $l_j \leftarrow \max\{k : k \leq \text{max_match} \wedge S[p - j : p - j + k - 1] = S[p : p + k - 1]\}$;
    if $l_j > l$ **then**
      $(l, d) \leftarrow (l_j, j)$;
  if $l \geq$ min_match **then**
    $t_m \leftarrow$ <Ll,Dd>;
    $p \leftarrow p + l$;
  else
    $t_m \leftarrow S[p]$;
    $p \leftarrow p + 1$;
return $(t_1, t_2, \ldots, t_m)$;

We illustrate the LZ77 hash table with a toy example. For simplicity, we assume the hash value of any string S is $S[1]$; for example, the hash value of `abc` is `a`.[10] Suppose we run LZ77 on input

```
draw drain train
```

and the pointer has just moved from `t` to `r` in `train`. A schematic view of the hash table at this point is shown in Fig. 1. We note that there are many possible ways to implement the hash table; in particular, it is not necessary to include the substrings themselves in the hash table, and we only do so for clarity.

hash value	a	d	i	n	r	t	w	␣
head	8	6	9	10	7	12	4	11

position	1	2	3	4	5	6	7	8	9	10	11	12
substring	dra	raw	aw␣	w␣d	␣dr	dra	rai	ain	in␣	n␣t	␣tr	tra
next	0	0	0	0	0	1	2	3	0	0	5	0

Fig. 1: A schematic view of the LZ77 hash table containing positions in the input `draw drain train` up to `t`. For better clarity, we also use "␣" to represent a space character. Here "head" refers to the first entry on the corresponding hash chain, and "next" refers to the next entry on the chain, or the end of the chain if it is 0.

At the current pointer position, LZ77 computes the hash value of `rai`, which is `r`. Next,[11] LZ77 traverses the corresponding hash chain, checking the positions of `rai` in `drain` and `raw` in `draw` for a match. It finds a match of 4 bytes starting at `r` in `drain` and outputs a back-reference <L4,D6>.

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

```
draw drain train
```

[10] Or, more precisely, the integer representation of `a`.
[11] Either before or after inserting the current position into the hash table, depending on the implementation.

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. LZ77 tries to find a longer match starting from the byte just after the pointer, and, if such a match is found, it discards the current match and takes the longer new match. More specifically, it outputs the current byte and advances the pointer by one byte, possibly starting lazy matching again on the new match. Otherwise, LZ77 proceeds as usual; that is, it outputs the back-reference for the current match and advances the pointer past it.

For example, running LZ77 on input

```
draw train drain
```

gives us

```
draw train <L3,D11>in
```

without lazy matching, and

```
draw train d<L4,D5>
```

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 [Huf52] is a well-known method for data compression. Huffman coding generates a prefix-free code called *Huffman code* according to the frequencies of the symbols, assigning shorter codewords for symbols that occur more frequently, to minimise the expected codeword 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 codeword.

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. [GHP13]). Here we only highlight some parts of Huffman coding that are relevant to our attacks in Sections 6 to 8. Note that, unlike LZ77, which is oblivious to the block structure in DEFLATE, Huffman coding in DEFLATE operates on blocks of data, and Huffman codes for each block do not depend on other blocks. The DEFLATE specification does not specify how to split the input data into blocks.

DEFLATE uses two Huffman codes to encode a block of the LZ77 output, with one Huffman code for both literals (i.e. unmatched bytes) and match lengths, and another Huffman code for match distances. DEFLATE decompressors can always determine which Huffman code to use, because a codeword in the LZ77 output represents a match distance if and only if its previous codeword represents a match length. Each Huffman code is selected subject to some additional requirements, such that one can recover a Huffman code solely from the lengths of the codewords; for example, shorter codewords should lexicographically precede longer codewords [Deu96a, Section 3.2.2]. The lengths of the Huffman codewords are measured in bits.

For Huffman coding, literals take symbols from 0 to 255 in the first Huffman code, the same as their integer value. 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 symbols from 257 to 285 in the first Huffman code, and distances take symbols from 0 to 29 in the second Huffman code.[12] To differentiate between different lengths or distances with the same symbol, DEFLATE appends zero or more extra bits to each length or distance symbol. The exact number of extra bits depends on the symbol: for example, a distance of 101 bytes is encoded as bits $110100100_{(2)}$ in the LZ77 output, a

[12] The DEFLATE specification refers to length and distance symbols as length and distance codes [Deu96a, Section 3.2.5], whereas we use symbols instead of codes in order to avoid confusion with Huffman codes.

concatenation of the distance symbol $1101_{(2)} = 13$ for distances 97–128 and five extra bits $00100_{(2)} = 4$ to indicate an exact distance of 101. Note that the extra bits do not participate in Huffman coding and therefore stay unchanged in the final output.

Table 1 shows the symbols and extra bits for distance encoding in DEFLATE. Note that shorter distances generally require fewer extra bits, the rationale being that closer repetitions tend to occur more frequently. The match length encoding in DEFLATE follows a similar logic, generally requiring fewer extra bits to encode shorter lengths; we refer interested readers to [Deu96a, Section 3.2.5].

Symbol	Extra Bits	Distance	Symbol	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 1: Symbols and extra bits for distance encoding in DEFLATE. [Deu96a, Section 3.2.5]

A DEFLATE implementation may choose whether to use fixed or dynamic Huffman codes for each block by comparing which option yields a better compression ratio, and a block can also be marked as uncompressed. If dynamic Huffman codes are used, then the final output also includes the length sequences of the Huffman codes for encoding the LZ77 output, compressed first with run-length encoding and then with another Huffman code. As with most existing attacks on compression,[13] our attack techniques work for both fixed and dynamic Huffman codes.

3.4 zlib

For a concrete implementation of DEFLATE, we look at zlib [RGA], a compression library that implements DEFLATE compressors and decompressors. zlib is the only DEFLATE-compliant implementation provided in the DEFLATE specification [Deu96a, Section 7]. It is widely used to produce compressed data in zlib, DEFLATE or gzip format. Our description of zlib is based on zlib version 1.3.2,[14] but it likely also applies to most 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; our description here also applies to GNU Gzip. Previous works often conflate DEFLATE with zlib/Gzip.

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.

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 in the sliding window that represent substrings of min_match bytes. The zlib implementation of LZ77 uses an unkeyed "rolling hash", for which we give more details later. At each new position, LZ77 in zlib

[13] In fact, some existing attacks may work more reliably when the Huffman codes are fixed, and the EFAIL attack [PDM+18] actually requires the Huffman codes to be fixed, albeit in a different context.

[14] https://github.com/madler/zlib/releases/tag/v1.3.2

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 a compressed data format.

In deflate_fast, LZ77 does not perform lazy matching and does not insert positions within a matched substring that is strictly longer than max_insert bytes into the hash table, with the exception that the first position of a matched substring is always inserted into the hash table. Note that max_insert is not used in the default deflate_slow mode and is typically 4-6 in deflate_fast; see Table 2.

In deflate_slow, LZ77 adds all positions in the sliding window to the hash table, and it performs lazy matching when the match at the current position 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 $\lfloor \mathsf{max_chain}/4 \rfloor$ entries in the hash table during lazy matching.

We note that LZ77 in zlib cannot create back-references to a substring that starts at the first byte of the sliding window,[15] and therefore cannot match a substring starting at the first byte of the input. This fact is generally irrelevant for compression side-channel attacks in practice, but might be useful to know for readers who wish to test our techniques in an idealised environment.

Hash function. The hash function in zlib, $\mathsf{H}_{\mathrm{zlib}}$, is defined as follows.[16] Let S be an arbitrary byte string of length min_match (3 by default); then

$$\mathsf{H}_{\mathrm{zlib}}(S, \mathsf{hash_bits}) := \left(\bigoplus_{i=1}^{\mathsf{min_match}} S[i] \cdot 2^{(\mathsf{min_match}-i)\cdot \left\lceil \frac{\mathsf{hash_bits}}{\mathsf{min_match}} \right\rceil} \right) \bmod 2^{\mathsf{hash_bits}},$$

where hash_bits is 15 by default and can be at most 16.[17]

Observe that, for every byte $b \in \{0, \ldots, 255\}$, we have

$$\begin{aligned}
&\mathsf{H}_{\mathrm{zlib}}(S\,[2 : \mathsf{min_match}] \,\|\, b, \mathsf{hash_bits}) \\
&= \left(b + \bigoplus_{i=2}^{\mathsf{min_match}} S[i] \cdot 2^{(\mathsf{min_match}-i+1)\cdot \left\lceil \frac{\mathsf{hash_bits}}{\mathsf{min_match}} \right\rceil} \right) \bmod 2^{\mathsf{hash_bits}} \\
&= \left(b + 2^{\left\lceil \frac{\mathsf{hash_bits}}{\mathsf{min_match}} \right\rceil} \cdot \mathsf{H}_{\mathrm{zlib}}(S, \mathsf{hash_bits}) - S[1] \cdot 2^{\mathsf{min_match}\cdot \left\lceil \frac{\mathsf{hash_bits}}{\mathsf{min_match}} \right\rceil} \right) \bmod 2^{\mathsf{hash_bits}} \\
&= \left(b + 2^{\left\lceil \frac{\mathsf{hash_bits}}{\mathsf{min_match}} \right\rceil} \cdot \mathsf{H}_{\mathrm{zlib}}(S, \mathsf{hash_bits}) \right) \bmod 2^{\mathsf{hash_bits}},
\end{aligned}$$

where the last equation holds because $2^{\mathsf{hash_bits}}$ divides $2^{\mathsf{min_match}\cdot\lceil \mathsf{hash_bits}/\mathsf{min_match} \rceil}$, which, in turn, is because $\mathsf{min_match} \cdot \lceil \mathsf{hash_bits}/\mathsf{min_match} \rceil \geq \mathsf{hash_bits}$. Therefore, the hash value at the next position of the input only depends on the current hash value and the next byte, which means LZ77 does not need to compute the new hash value from scratch. This property of the hash function is known as *rolling*; for a formal definition, see Definition 2 in Section 7.1.

In the default setting, for a 3-byte string abc, we have

$$\mathsf{H}_{\mathrm{zlib}}(abc) = \big((a \cdot 2^{10}) \oplus (b \cdot 2^5) \oplus c \big) \bmod 2^{15},$$

where we omit the parameter $\mathsf{hash_bits} = 15$ for simplicity.

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 as the compression level increases.

[15] This is a consequence of simplifying the compressor implementation; see e.g. https://github.com/madler/zlib/blob/v1.3.2/deflate.c#L1398-L1399.

[16] See https://github.com/madler/zlib/blob/v1.3.2/deflate.c#L141. Note that zlib only defines the function to update the hash value at the next position as we will give later, but the two definitions are clearly equivalent.

[17] Except for 16-bit machines, which are rarely used today and are therefore out of the scope of this paper.

Table 2 shows the configuration for each compression level in zlib.[18] zlib does not perform compression at level 0. At levels 1 to 3, zlib uses the deflate_fast compression mode, which does not insert substring positions in a match above max_insert bytes into the hash table; at 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	-	-	-	-	-	deflate_stored
1	-	-	4	8	4	deflate_fast
2	-	-	5	16	8	deflate_fast
3	-	-	6	32	32	deflate_fast
4	4	4	-	16	16	deflate_slow
5	8	16	-	32	32	deflate_slow
6	**8**	**16**	**-**	**128**	**128**	deflate_slow
7	8	32	-	128	256	deflate_slow
8	32	128	-	258	1024	deflate_slow
9	32	258	-	258	4096	deflate_slow

Table 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 configuration in bold.

3.5 DEFLATE implementation zoo

While zlib is colloquially considered to be the de facto reference implementation of DEFLATE, many other DEFLATE implementations are also available, and some are expected to gain a wider adoption in the future. As applications are generally oblivious to the internals of the compressors they depend on, the implementation details of DEFLATE compressors can be very different, and it is relatively easy for developers to replace zlib/Gzip with another DEFLATE-compatible compressor in an application.

We can roughly split alternative DEFLATE implementations into the following categories:

1. Forks of zlib/Gzip to improve compression quality and/or speed, e.g. zlib-ng.
2. zlib equivalents in different programming languages, e.g. flate/zlib in Go.
3. Alternative DEFLATE implementations. These include historical compressors like PKZIP [PKW], for which Phil Katz created the DEFLATE format, or simply reimplementations of DEFLATE for patent, performance, portability or other reasons, like miniz.

It is perhaps surprising that DEFLATE compressors use a plethora of different hash functions, as we will show below. The possible reasons include:

1. The hash function is invoked many times (roughly the same as the input size in bytes) during compression, so having a faster hash function implementation can improve compression speed.
2. A good hash function reduces the number of collisions in the hash table, thereby improving compression speed by reducing the number of hash chain entries to look up each time, and potentially improving compression quality by a small margin.
3. Compared to other parts of DEFLATE, the hash function, typically consisting of no more than a few lines of code, is very easy to tweak.

In the remainder of this section, we examine some notable alternative DEFLATE implementations, and we highlight their differences to zlib that are relevant to compression side-channel attacks. Note that we limit our discussion to little-endian architecture. Our study does not include all DEFLATE implementations; for example, we exclude libdeflate[19] and fflate.[20] However, with few to no modifications, our techniques can be applied to all compressor implementations that do not significantly deviate from the DEFLATE algorithm.

[18] Table reproduced from the zlib source code; see https://github.com/madler/zlib/blob/v1.3.2/deflate.c#L112-L124.

[19] https://github.com/ebiggers/libdeflate/

[20] https://101arrowz.github.io/fflate/

zlib-ng. zlib-ng [zli] (where "ng" stands for "next generation") is a compression library designed as a drop-in replacement for zlib. zlib-ng is based on zlib, on top of which the developers implemented many software and hardware optimisations, some taken from other DEFLATE implementations. Our description is based on version 2.3.3 of zlib-ng.[21]

We briefly summarise the changes zlib-ng made that have the most impact on its compressor output:

1. zlib-ng increases the default for min_match from 3 to 4[22] at compression levels 1-8.[23]
2. zlib-ng supports two new compression strategies, deflate_quick (at level 1) and deflate_medium (at levels 3-6, which means deflate_medium is the default compression strategy in zlib-ng), both based on DEFLATE, which can be disabled[24] by adjusting the relevant build option.[25] Roughly speaking, the deflate_quick strategy can be seen as the deflate_fast strategy with max_insert $<$ min_match and max_chain $= 1$, and the deflate_medium strategy can be seen as the deflate_fast strategy with a very large max_insert.
   Our techniques in Sections 6 and 8 also apply to deflate_quick and deflate_medium.[26] While our collision-based amplification technique and templates in Section 7 are also valid for deflate_medium, we need a separate construction for deflate_quick because of the extremely small max_chain, which we present in Section 7.4.4.
3. At level 9,[27] zlib-ng modifies the deflate_slow strategy to use a different method for lazy matching, which may traverse different hash chains than the one corresponding to the position after the LZ77 pointer.[28] As a consequence, while our techniques in Sections 6 and 8 are still valid at level 9, the constructions for collision-based amplification queries for deflate_slow in Section 7 as described do not apply to level 9 in zlib-ng.
4. At levels 1-8, zlib-ng uses a different hash function for the LZ77 hash table; it is the following multiplicative hash function.[29] For a 4-byte string S,

$$\mathsf{H}_{\text{zlib-ng}}(S) := \left\lfloor \frac{(\mathsf{uintl}(S) \cdot 2654435761) \bmod 2^{32}}{2^{16}} \right\rfloor,$$

   where $\mathsf{uintl}(S)$ is the integer representation of S in little-endian. Equivalently, for a 4-byte string $abcd$,

$$\mathsf{H}_{\text{zlib-ng}}(abcd) = \left\lfloor \frac{\left(\left(a + b \cdot 2^8 + c \cdot 2^{16} + d \cdot 2^{24}\right) \cdot 2654435761\right) \bmod 2^{32}}{2^{16}} \right\rfloor.$$

   The magic number 2654435761 is a prime number, often referred to as the golden ratio prime number, because it is very close to $2^{32} \cdot \varphi \approx 2654435769.5$, where $\varphi = (\sqrt{5} - 1)/2$ is the golden ratio. This hash function is an instance of Fibonacci hashing; see [Knu98, Section 6.4] for a description of multiplicative hashing schemes and an argument in favour of using Fibonacci hashing.
   At level 9, zlib-ng uses the same hash function as the default rolling hash function in zlib/Gzip.[30]
   We note that older versions of zlib-ng have used different hash functions depending on hardware; more specifically, zlib-ng may use CRC-32C on Intel CPUs supporting SSE4.2, and CRC-32 on ARM

[21] https://github.com/zlib-ng/zlib-ng/releases/tag/2.3.3

[22] DEFLATE implementations with an increased default min_match usually distinguish between the minimum allowed match length (of 3 bytes) in the DEFLATE standard and the minimum match length actually considered to be valid in the implementation. For simplicity, we do not make such distinctions in this paper, but rather just use min_match for the latter meaning.

[23] In fact, the situation is slightly more complicated for the deflate_slow strategy in zlib-ng: while deflate_slow in zlib-ng allows 3-byte matches (github.com/zlib-ng/zlib-ng/blob/2.3.3/deflate_slow.c#L77), the hash function in zlib-ng when max_chain ≤ 1024 (which corresponds to levels 1-8) takes 4-byte strings as input, essentially forcing matches to be at least 4 bytes.

[24] And these new strategies are indeed disabled for compatibility in the `zlib-ng-compat` package, which replaces zlib in Fedora; see https://koji.fedoraproject.org/koji/fileinfo?rpmID=46564084&filename=zlib-ng.spec.

[25] The deflate_quick and deflate_medium strategies were first introduced by the (now deprecated) Intel fork of zlib and subsequently ported to zlib-ng; see https://github.com/intel/zlib.

[26] Because the `fizzle_matches` procedure in deflate_medium may adjust the boundary between two neighbouring back-references in the LZ77 output, making the former shorter and the latter longer, the actual LZ77 output for our constructions may slightly differ from our descriptions. However, our constructions still achieve their intended effects.

[27] More precisely, if max_chain > 1024; see https://github.com/zlib-ng/zlib-ng/blob/2.3.3/deflate_slow.c#L24.

[28] This modification was ported from the fast_zlib project; see https://www.gildor.org/en/projects/zlib.

[29] https://github.com/zlib-ng/zlib-ng/blob/2.3.3/insert_string.c

[30] https://github.com/zlib-ng/zlib-ng/blob/2.3.3/insert_string_roll.c

CPUs supporting ACLE. These hash functions were removed from zlib-ng starting from version 2.2.0.[31]

Cloudflare fork of zlib. The Cloudflare fork of zlib [Clo], hereafter referred to as *cloudflare-zlib*, claimed to have made "massive performance improvements" [Clo] to the original zlib. It only supports 64-bit machines with Intel or ARM CPUs. Our description is based on version 0.3.6 of cloudflare-zlib.[32] Note that cloudflare-zlib has recently been deprecated in order to avoid using memory-unsafe languages.[33]

Most of the changes made in cloudflare-zlib focus on improving the compression speed rather than the compression ratio, meaning that they should be largely irrelevant to compression side-channel attacks. Two of the changes are notable exceptions:

1. Cloudflare-zlib increases the default for $\mathsf{min_match}$ from 3 to 4.
2. Cloudflare-zlib uses different hash functions for the LZ77 hash table. There are two hash functions in cloudflare-zlib.[34] For a 4-byte string S, if ARM ACLE or Intel SSE4.2 is supported, then the hash function is a truncated CRC-32C [CBH93]:

$$\mathsf{H}_{\text{Cloudflare}}(S, \mathsf{hash_bits}) := \mathsf{CRC\text{-}32C}(0, S) \bmod 2^{\mathsf{hash_bits}};$$

otherwise, cloudflare-zlib falls back to a generic multiply-xor hash function:

$$\mathsf{H}'_{\text{Cloudflare}}(S, \mathsf{hash_bits}) := \left((\mathsf{uintl}(S) \cdot 246822519) \oplus \left\lfloor \frac{(\mathsf{uintl}(S) \cdot 246822519) \bmod 2^{32}}{2^{19}} \right\rfloor \right) \bmod 2^{\mathsf{hash_bits}},$$

where the magic number 246822519 is a prime number taken from the xxHash[35] implementation. Same as in zlib/Gzip, we have $\mathsf{hash_bits} = 15$ by default in both cases.

Chromium fork of zlib. The Chromium fork of zlib [chr], hereafter referred to as *chromium-zlib* in short, modifies zlib to improve compression and decompression speed. In addition to Chromium-based browsers and application frameworks, chromium-zlib is used in Node.js,[36] a JavaScript runtime environment for network applications. Our description is based on version 1.3.2 of chromium-zlib.[37]

The main change in chromium-zlib relevant to our attacks is that it uses a different hash function for the LZ77 hash table. Chromium-zlib uses a multiplicative hash function defined as follows.[38] For a 4-byte string S,

$$\mathsf{H}_{\text{Chromium}}(S) := \left\lfloor \frac{(\mathsf{uintl}(S) \cdot 66521 + 66521) \bmod 2^{32}}{2^{16}} \right\rfloor \bmod 2^{\mathsf{hash_bits}}.$$

Equivalently, for a 4-byte string $abcd$,

$$\mathsf{H}_{\text{Chromium}}(S) := \left\lfloor \frac{\left(\left(a + b \cdot 2^8 + c \cdot 2^{16} + d \cdot 2^{24}\right) \cdot 66521 + 66521\right) \bmod 2^{32}}{2^{16}} \right\rfloor \bmod 2^{\mathsf{hash_bits}}.$$

The magic number 66521 is a composite number,[39] as $66521 = 7 \cdot 13 \cdot 17 \cdot 43$. Note that, even though $\mathsf{min_match} = 3$ in Chromium-zlib by default, the use of a hash function that operates on 4-byte strings makes it very difficult, if possible at all, for LZ77 in Chromium-zlib to find 3-byte matches.

[31] https://github.com/zlib-ng/zlib-ng/releases/tag/2.2.0
[32] https://github.com/cloudflare/zlib/commit/1252e2565573fe150897c9d8b44d3453396575ff
[33] https://github.com/cloudflare/zlib/blob/d9f746b9a0fb41b2c7d06ed199d14543acbb6275/README.md
[34] https://github.com/cloudflare/zlib/blob/1252e2565573fe150897c9d8b44d3453396575ff/deflate.c#L132
[35] https://xxhash.com/
[36] https://nodejs.org/
[37] https://chromium.googlesource.com/chromium/src/third_party/zlib/+/9ffe2c6b
[38] See https://chromium.googlesource.com/chromium/src/third_party/zlib/+/9ffe2c6b/contrib/optimizations/insert_string.h. We assume that the build option `USE_ZLIB_RABIN_KARP_HASH` is disabled, and it is indeed disabled by default. Older versions of Chromium-zlib may use a truncated CRC-32C as the hash function.
[39] Readers should not confuse the magic number 66521 with the modulus 65521 in the Adler-32 checksum algorithm; the latter is the largest prime number below 2^{16}.

zlib in Go. The standard library of Go provides the functionalities of zlib through three packages: zlib,[40] gzip[41] and flate.[42] The packages zlib and gzip are essentially wrappers over the flate package, while the latter is an implementation of DEFLATE in Go. We refer to these packages collectively as flate/zlib for clarity. Our description is based on go1.25.9, but it likely applies to most versions of Go.

flate/zlib is not merely a port of zlib from C to Go; instead, it differs from zlib in many details. We briefly summarise the differences between flate and zlib that are relevant to our attacks:

1. flate/zlib hard-codes most DEFLATE parameters, only leaving the compression level adjustable, and increases min_match from 3 to 4.
2. At compression level 1, flate/zlib implements a custom algorithm based on the LZ77-style encoder in Snappy [Sna], a compressor developed by Google that focusses on speed. This compression algorithm is very different from DEFLATE and is therefore out of the scope of this paper. However, it is clearly susceptible to simple variants of our amplification techniques in Sections 6 and 7.[43]
3. flate/zlib supports the use of preset dictionaries for compression and decompression.
4. flate/zlib uses different hash functions for the LZ77 table.
   At compression levels 2-9, flate/zlib hard-codes hash_bits to 17 and performs multiplicative hashing on the big-endian representation of a 4-byte string S;[44] more specifically,

$$\mathsf{H}_{\mathrm{Go}}(S) := \left\lfloor \frac{(\mathsf{uintb}(S) \cdot 506832829) \bmod 2^{32}}{2^{15}} \right\rfloor.$$

   Equivalently, for a 4-byte string $abcd$,

$$\mathsf{H}_{\mathrm{Go}}(abcd) = \left\lfloor \frac{\left((a \cdot 2^{24} + b \cdot 2^{16} + c \cdot 2^{8} + d) \cdot 506832829\right) \bmod 2^{32}}{2^{15}} \right\rfloor.$$

   The magic number 506832829, contrary to what some might expect, is actually a composite number, as $506832829 = 397 \cdot 1276657$. This number also appears in the hash functions of some other compressors developed by Google, e.g. Snappy and brotli [bro].
   At compression level 1, flate/zlib employs a similar hash function as above,[45] except for using the little-endian representation and lowering hash_bits to 14: for every string S of length 4,

$$\mathsf{H}_{\mathrm{Gofast}}(S) := \left\lfloor \frac{(\mathsf{uintl}(S) \cdot 506832829) \bmod 2^{32}}{2^{18}} \right\rfloor.$$

Miniz. Miniz [min] is a DEFLATE implementation in C, notable for its portability. A miniz port to Rust, miniz_oxide,[46] is the default backend of flate2,[47] the most widely used compressor in Rust. Our description is based on version 3.1.1 of miniz.[48]

While the codebase of miniz is independent from zlib, their implementation details share many similarities, in particular using the same default min_match and the same hash function in LZ77. The main differences between miniz and zlib that are of interest are:

1. In addition to compression levels 0-9, miniz introduces level 10 (denoted as "`UBER_COMPRESSION`"), which uses the deflate_slow strategy with max_chain = 1500, the largest among all levels in miniz.
2. Many compression parameters in zlib are either unused or different in miniz. More specifically, max_insert is not used in miniz, max_lazy is hard-coded to 128 at levels 4-10 and unused at other levels, nice_length is hard-coded to 128 at all levels, good_length is hard-coded to 32 at all levels (and with a slightly different meaning), and max_chain takes different values.

[40] https://pkg.go.dev/compress/zlib@go1.25.9
[41] https://pkg.go.dev/compress/gzip@go1.25.9
[42] https://pkg.go.dev/compress/flate@go1.25.9
[43] A main limitation is that the LZ77-style encoder is sensitive to DEFLATE block boundaries, so our queries may not work as desired if they span more than one block.
[44] https://cs.opensource.google/go/go/+/refs/tags/go1.25.9:src/compress/flate/deflate.go;l=296
[45] https://cs.opensource.google/go/go/+/refs/tags/go1.25.9:src/compress/flate/deflatefast.go;l=37
[46] https://crates.io/crates/miniz_oxide/0.9.0
[47] https://crates.io/crates/flate2/1.1.10
[48] https://github.com/richgel999/miniz/releases/tag/3.1.1

3. Due to a particularity of its hash table design, LZ77 in miniz discards matches starting at position $2^{15} \cdot n + 1$ in the plaintext for every $n \in \mathbb{N}$;[49] moreover, LZ77 in miniz stops looking for a match as soon as having encountered such a position on the hash chain.[50] The constant 2^{15} is independent from the sliding window size window_size. As a consequence, our techniques in Sections 6 to 8 might not work as desired if the constructed queries contain such positions. In particular, most of our templates for collision-based amplification in Sections 7.3 and 7.4 cannot achieve unlimited amplification for miniz. We will discuss more about this limitation in Section 7.3.6.
4. LZ77 in miniz uses a slightly different strategy for finding matches.[51] We do not go into the details here, but we remark that LZ77 may stop looking for a match after checking slightly fewer than max_chain entries on the hash chain, as the maximum number of entries to check may vary according to the entries already checked. This complication affects the bounds in Sections 7.3 and 7.4 for our collision-based amplification technique, but it has no impact in practice except in rare cases.
5. At level 1, miniz may use a fast compression algorithm that is apparently based on LZRW1 [Wil91].[52] This compression algorithm is very different from LZ77 in DEFLATE and is therefore out of the scope of this paper.

4 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. Similar techniques can be used for testing whether the secret is equal to a particular value. We abstract away the technical details of the targeted protocols and applications, and we generally exclude techniques that make compression side-channel attacks work against a specific target (e.g. two canaries for Threema [PST23, Appendix B] and compressibility scores for databases [HME23, Section V]).

The attacks in this section all assume access to a compression length oracle. The exact form of the compression length oracle varies; generally speaking, the oracle takes a query and returns the compressed length of some string that contains, possibly among other data, the query and a fixed unknown secret. In practice, the oracle is realised through observing the length of the ciphertext.[53] The compressor implements the DEFLATE algorithm. Unless otherwise stated, the attacks are adaptive, i.e. subsequent queries may depend on oracle responses to previous queries. The goal of the attacker is to recover the secret (with some adequate probability) and, simultaneously, minimise the number of oracle queries, making the attack more practical and less prone to detection.

4.1 Kelsey's secret extraction attacks

Kelsey [Kel02] proposed two compression side-channel attacks for extracting a secret in the plaintext. In Kelsey's attack model, an attacker can make n oracle queries $Q_1, \ldots, Q_n$ and learn the DEFLATE-compressed length of $Q_i \| S$ for each $i \in [n]$, where S is an unknown secret. The goal of the attacker is to extract the secret S, while minimising the number of queries n used in the attack. Let Ω denote the alphabet of the secret.

Adaptive chosen-input attack. An adaptive attacker can construct a query Q as

$$Q = \mathsf{prefix} \| \mathsf{guess} \| \mathsf{filler} \| \mathsf{prefix},$$

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

As Kelsey explained, if the guess is correct, then $Q \| S$ contains a longer repeated substring prefix‖guess and its compressed length should be smaller than in the case where the guess is incorrect. After finding

[49] https://github.com/richgel999/miniz/blob/3.1.1/miniz_tdef.c#L1181
[50] https://github.com/richgel999/miniz/blob/3.1.1/miniz_tdef.c#L846
[51] See the definition of function `tdefl_find_match` at https://github.com/richgel999/miniz/blob/3.1.1/miniz_tdef.c#L829.
[52] In more detail, miniz calls `tdefl_compress_fast` (defined at https://github.com/richgel999/miniz/blob/3.1.1/miniz_tdef.c#L885) instead of `tdefl_compress_normal` if both the parameter `num_probes` is set to 1 (which is true for level 1 in miniz) and the option `MINIZ_USE_UNALIGNED_LOADS_AND_STORES` is enabled.
[53] For simplicity, we assume that the attacker can always efficiently recover the exact length of the compressed plaintext from the ciphertext, but we note that it is not necessary for the existing techniques to work.

the correct guess, the attacker can extend the first prefix in Q to prefix$\|$guess and move on to guess the next k characters of S, continuing 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 queries is in theory $(\ell|\Omega|^k)/(2k)$. Kelsey implemented the attack against zlib and observed that the attack was quite unstable, requiring a relatively large k and routine manual backtracking [Kel02, Section 6.1].

Non-adaptive chosen-input attack. A non-adaptive attacker can set $Q_1, \ldots, Q_n$ to all possible strings of a certain length $k < [|S|]$ and, for each $i \in [n]$, estimate the likelihood of Q_i being a substring in S from the compressed length of $Q_i\|S$; the attacker can then recover the secret by piecing together queries that are most likely to appear in S.

If S is sampled from Ω^ℓ, then the attacker needs to make $n = |\Omega|^k$ queries. While Kelsey did not specify how to choose k, it is clear that k should be large enough to counter the effect of noise and decrease the number of candidates that are consistent with the oracle responses. Kelsey implemented the attack against zlib. For a secret S chosen uniformly at random from $\{\texttt{0}, \ldots, \texttt{9}\}^{16}$, setting $k = 5$,[54] the attack returns a list of 20 candidates that contains S with probability around 70% [Kel02, Section 6.2].

4.2 CRIME/BREACH

In CRIME [RD12] and BREACH [GHP13], the goal of an attacker is to recover a secret cookie ck from the plaintext using a compression length oracle. The cookie ck is a byte string chosen uniformly at random from Ω^ℓ, and it always follows a known prefix P (e.g. `secret=`) in the plaintext. The attacker can make adaptive queries to the compression length oracle and observe the compressed length of the plaintext, in which the query is co-located with the prefix and the secret cookie. In other words, upon a query Q made by the attacker, the oracle returns the compressed length of the plaintext, which is either of the form $\cdots\|Q\|\cdots\|P\|ck\|\cdots$ or $\cdots\|P\|ck\|\cdots\|Q\|\cdots$; the fields marked by $\cdots$ are unspecified and only change minimally between queries.

Generally, both CRIME and BREACH attacks recover the secret cookie ck byte-by-byte, similar to Kelsey's adaptive chosen-input attack [Kel02]. 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 byte in ck, an attacker iterates over the alphabet Ω, making two specially crafted queries Q_1 and Q_2 to the compression length oracle for each guess g, with the query Q_2 being a permutation of Q_1. If the oracle responses for the two queries differ, then the attacker regards g as the correct guess, appends g to the known prefix, and proceeds to recover the next unknown byte of ck in the same way, continuing this process 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. Assume that the query is located before the prefix and the secret cookie $P\|ck$ in the plaintext. The Two Tries method in CRIME makes two queries

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

for a guess g. Here filler and padding are two byte strings such that, when the LZ77 pointer reaches $P\|ck$ in the plaintext corresponding to the query, the substring $P\|g$ in Q_1 is within the sliding window, but $P\|g$ in Q_2 is outside of the sliding window.

According to Rizzo and Duong [RD12], 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 $P\|ck$ 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 short prefix P, preferably of length min_match $- 1$, or that P should appear in padding or elsewhere in the sliding window, as otherwise LZ77 can still find a match for the prefix P in Q_1 but not in Q_2 if the guess is incorrect, creating a (smaller) difference in oracle responses.

[54] Inferred from [Kel02, Section 6.2].

Two Tries in BREACH. The BREACH 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 $P\|ck$ in the plaintext, and $\mathsf{padding}$ is a short byte string that does not contain any byte in Ω.

If the guess g is correct, then Q_1 contains a longer repeated substring $P\|g$, so the oracle response for Q_1 is likely smaller than that of Q_2; otherwise, if the guess is incorrect, then both Q_1 and Q_2 likely contain 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* [GHP13, Section 2.4]. However, the attack may still miss the correct guess or identify multiple candidates for an unknown byte.

16K-1. Rizzo and Duong [RD12] described an attack technique called 16K-1, which can reduce the average number of queries needed by an attacker to around $\ell\lceil\log|\Omega|\rceil$. However, the attack leverages a feature in the TLS protocol to control the whole plaintext except for a single secret byte and does not apply to compression side channels in general.

Assume that the compression length oracle splits the plaintext of the form $Q\|P\|ck\|\cdots$ into records of 16KiB (except for the last record), compresses each record separately,[55] and returns the compressed length of each record. This is a simplified view of the compression side channel in TLS. Assume also that the attacker knows the exact implementation and configuration of the compressor.

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

1. The attacker makes a query Q of around 16KiB to the compression length oracle, such that the plaintext is split into two records, each compressed separately, and the only unknown byte in the first record is $ck[1]$.
2. On receiving the oracle response, the attacker checks offline every guess $g \in \mathsf{cands}$, only keeping g in cands if substituting the unknown byte in the first record with g yields another record that compresses to the same length.

After that, the attacker regards the only element in cands as the unknown byte in ck, and proceeds to recover the rest of the cookie ck byte-by-byte in similar ways.

It is clear that the 16K-1 attacker never returns an incorrect cookie (under ideal conditions). The number of queries performed depends on the construction of the query Q, which Rizzo and Duong did not specify. One possibility is that Q is constructed as the repeated concatenation of half of the values in cands, so that Huffman coding in DEFLATE encodes the LZ77 output of the record to a slightly shorter length if the unknown byte falls within that half of cands. Each query eliminates around half of the values in cands, similar to a binary search.

Rizzo further claimed[56] that 16K-1 can be fine-tuned to use as few as 4 queries to recover an unknown byte when presumably $|\Omega| = 64$, which may require a different construction of the query Q than what we have described, e.g. sampling each byte of Q according to a skewed distribution on cands.

Divide and conquer. Despite not having been covered in [RD12, GHP13], 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 technique was likely introduced by Pornin and later rediscovered in [KZ16] (see Section 2.2).

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

$$P\|g_1\|\mathsf{filler}_1\|P\|g_2\|\mathsf{filler}_2\|\cdots\|P\|g_{\lfloor m/2\rfloor-1}\|\mathsf{filler}_{\lfloor m/2\rfloor-1}\|P\|g_{\lfloor m/2\rfloor}$$

[55] In fact, depending on the implementation, LZ77 may still be able to find back-references in previous records, but the first record is always compressed separately since it does not have any previous record.

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

where $\mathsf{filler}_1, \ldots, \mathsf{filler}_{\lfloor m/2 \rfloor - 1}$ are optional fillers to separate different guesses. If the first unknown byte in ck is in $\{g_1, \ldots, g_{\lfloor m/2 \rfloor}\}$, then Q contains a repeated substring $P \| g_i$ 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 compressed length is shorter, and to $\{g_{\lfloor m/2 \rfloor + 1}, \ldots, g_m\}$ if the compressed length is longer. In practice, the attacker may need to use a variant of the Two Tries method to make the signal visible. The average number of queries is in theory around $\ell \lceil \log |\Omega| \rceil$.

The divide-and-conquer technique can potentially be used to improve many compression side-channel attacks. However, this technique may suffer from greater instability,[57] which partly offsets the reduced number of queries.

5 Attack model and assumptions

In this section, we present the model for our attacks and outline our assumptions. Our attack model abstracts the "CRIME-style" attacks of e.g. [RD12, GHP13], where an attacker tries to recover a secret by adaptively querying a compression length oracle that returns the compressed length of some plaintext that contains both the secret and the query.[58]

Our attack model is similar to the cookie recovery (CR) security game defined in [ASB15]. However, the attacker in our model has weaker capabilities, as the attacker only knows a short prefix before the cookie and controls a part of the plaintext located after the cookie, while other parts of the plaintext may be unknown to the attacker and may change on each query, ruling out attacks that require full control or knowledge of the plaintext and allowing noise.

5.1 Attack model

Let Σ and Ω be two alphabets such that $\Omega \subseteq \Sigma \subseteq \{0, \ldots, 255\}$. Let $\ell, \ell_P, L_Q \in \mathbb{N}_+$. A target possesses a secret $ck \in \Omega^\ell$ (ck for "cookie"), and an attacker tries to recover ck through a compression side channel, abstracted as a compression length oracle. The attacker can submit queries $Q \in \Sigma^*$ of length $\ell_Q := |Q|$, where $\ell_Q \leq L_Q$; the oracle responds to each query Q with the compressed length of a plaintext embedded with both the secret and the query. The attacker can adaptively query the compression length oracle. The attacker wins if they are able to recover the secret ck.

The compression length oracle returns

$$|\mathsf{C}(\cdots \| P \| ck \| \cdots \| Q \| \cdots)|,$$

where C is a compressor and P is a prefix before the secret, which we will elaborate on below. The parts represented by $\cdots$ in the plaintext are unknown to the attacker and may change with each query.

The compressor C implements the DEFLATE algorithm described in Section 3.3. While compressors are usually deterministic, we allow C to be probabilistic to account for e.g. random padding [Deg21, PFSG22] and differentially-private compressors [BLG25]. Unless otherwise specified, we assume that C represents the zlib compressor (Section 3.4), but our techniques and analysis apply to most DEFLATE compressors. In particular, in our analysis, we may give the validity range of the parameters in our techniques based on parameters in zlib that are not present in the DEFLATE specification, such as $\mathsf{max_chain}$, $\mathsf{nice_length}$, $\mathsf{max_insert}$ and $\mathsf{max_lazy}$, but these parameters are shared by most compressors we examined in Section 3.5. We assume the compressor configuration is valid, in that every parameter of the compressor has a reasonable value; in particular, we assume $\mathsf{min_match} \geq 2$,[59] $\mathsf{nice_length} \geq \mathsf{min_match}$, $\mathsf{min_match} \leq \mathsf{max_insert} < \mathsf{max_match}$ in $\mathsf{deflate_fast}$, and $\mathsf{min_match} < \mathsf{max_lazy} \leq \mathsf{max_match}$ in $\mathsf{deflate_slow}$.

The prefix $P \in \Sigma^{\ell_P}$ is a byte string located before the secret. Both the prefix P and the secret ck stay unchanged across different queries. We assume that $\ell_P \geq \mathsf{min_match}$, where $\mathsf{min_match}$ is the minimum

[57] Generally speaking, a more complex query is more likely to result in unwanted behaviours of DEFLATE. For example, it is possible that, because the prefix P appears around $\lfloor m/2 \rfloor$ times in the plaintext, Huffman coding in DEFLATE encodes the back-reference for P more efficiently, making the compressed length difference of the two cases smaller.

[58] This is an idealised representation of real-world compression length oracles that reveal only the *ciphertext* length, which may in turn be affected by noise due to added random padding or padding from the block cipher mode of operation. We show in Section 9 that our techniques work also for such oracles.

[59] We note that the case of $\mathsf{min_match} = 2$ is only of theoretical interest, because the DEFLATE specification requires that the length of any back-reference be at least 3 bytes [Deu96a, Section 3.2.5].

match length of LZ77 in the compressor, but $\ell_P \geq \mathsf{min_match} - 1$ is usually sufficient in practice. To simplify our discussion, we assume that $\ell_P < \mathsf{max_match}$, where $\mathsf{max_match}$ is the maximum match length; note that the attacker can use a suffix of P for the attack instead if P is too long.

We require that the query Q be close to the prefix and the secret $P\|ck$ in the plaintext, and in particular their distance should be smaller than the sliding window size $\mathsf{window_size}$; note that we do not require that the entire query be in the same sliding window as the secret.

For our description of the amplification techniques in Sections 6 and 7, we focus on the case where the attacker tries to recover the first byte of the secret, $ck[1]$, because the attacker can recover the entire secret byte-by-byte, as described in Section 4. In more detail, after recovering the first $\ell' < \ell$ bytes of the secret, the attacker can regard $P\|ck\,[1:\ell']$ as the known prefix and $ck\,[\ell':\ell]$ as the secret, after which the attacker can proceed to recover $ck[\ell'+1]$ using the same method for recovering $ck[1]$; by repeating this process, the attacker can eventually recover the entire secret.

It is clearly possible to make certain changes 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. Our attack model is not intended to be complete, but rather intended to be broad enough to show the power of the framework while remaining conceptually simple to understand.

5.2 Assumptions

In addition to the assumptions in Section 5.1 that are generally shared by existing compression side-channel attacks, our new techniques in Sections 6 to 8 may rely on the following special assumptions. While these assumptions are implicitly present in [ASB15] and hold for many compression side channels in practice, they are not generally true for all compression side channels.

Positions of query and secret. We assume the query Q is embedded after the secret ck, but not necessarily directly; i.e. the index of ck is smaller than the index of Q in the plaintext. This assumption is already present in our attack model, but it is generally not required for existing compression side-channel attacks; in fact, the 16K-1 technique in [RD12] only works if the query is located *before* the secret. Unfortunately, this assumption is crucial for almost all of our techniques.

Nevertheless, we expect that it should be feasible for queries to be located after the secret in compression side channels. Indeed, many existing compression side-channel attacks were based on such queries, e.g. [HME23, PST23]. For web servers vulnerable to BREACH and its variants, an example where this assumption holds was given in [PFSG22]. While CRIME exploited the request path being located before the secret cookie in HTTP requests, it is clearly possible to launch a similar attack using control of the request body located after the secret cookie.

Knowledge of compressor. We assume that the attacker knows the compressor C used by the target, including its configuration (e.g. compression level). This assumption is important for some of our techniques, such as collision-based amplification (Section 7).

For existing compression side-channel attacks, the effect of different compressor implementations and configurations remains unstudied to our knowledge, and it is unclear whether the heuristics introduced in previous attacks work for all DEFLATE compressor implementations and configurations. The 16K-1 technique in CRIME [RD12] also assumes full knowledge of the compressor, as the attacker needs to simulate the compression length oracle locally.

Our assumption is consistent with Kerckhoffs' principle, as compressor implementations and configurations are generally not kept as secret; for example, they are publicly available in most open-source projects. Moreover, the attacker can obtain knowledge of the compressor using a fingerprinting attack. We refrain from diving into details but merely sketch this possibility here: in the most basic form, the attacker can deduce the compression level by estimating the compression ratio; a more sophisticated attacker can send crafted queries to the compression length oracle, whose responses can then be used to distinguish different compressors or different configurations, using variants of our techniques in the following sections.

Maximum query length. We assume that the maximum length of a query L_Q is not too small; we quantify this more precisely in our experiments. We will later show that longer queries are more powerful. In Sections 6 and 7, L_Q determines the maximum difference in compressed lengths between correct and incorrect guesses. In Section 8, we introduce techniques to reduce the number of queries by using longer queries.

Several existing techniques for compression side-channel attacks already used long queries: the Two Tries technique in CRIME [RD12] needs to fill the larger part of a sliding window of typically around 32KiB; the 16K-1 technique in CRIME [RD12], as the name suggests, utilises queries of around 16KiB; in the divide-and-conquer technique, the length of a query can reach hundreds of bytes when the alphabet Ω is large; DBREACH [HME23] requires filling a 4KiB page with attacker-controlled data to compute the compressibility of a guess.

5.3 Running example

We use the following example setting to illustrate our techniques in Sections 6 to 8. Let Σ be the set of ASCII-printable bytes, and let $\Omega \subseteq \Sigma$ be the set of ASCII digits, i.e. $\Omega = \{\texttt{0}, \ldots, \texttt{9}\} = \{48, \ldots, 57\}$. The secret ck is a byte string sampled according to some fixed distribution on Ω^ℓ. The target exposes a compression length oracle to an attacker, where the target uses zlib to compress a byte string

$$\cdots \| \texttt{secret=} \| ck \| \cdots \| Q \| \cdots$$

for each query $Q \in \Sigma^*$ of length $\ell_Q \leq L_Q$ to the oracle, and returns the length of the compressed byte string to the attacker. The attacker knows the compression level and parameters used by the target's compressor as well as the prefix `secret=`, which, except in the query Q, does not occur elsewhere in the plaintext. However, the attacker does not know the parts represented by $\cdots$, which may also change with each query. The goal of the attacker is to recover the secret ck by making adaptive queries to the oracle.

6 General amplification

In this section, we present some novel techniques to reduce or remove noise in compression side-channel attacks against DEFLATE by amplifying differences in compressed lengths with crafted queries. We design queries that make LZ77 outputs differ substantially depending on the secret, in ways such that differences in LZ77 outputs lead to length differences in DEFLATE-compressed data. Unlike existing proposals to mitigate noise using statistical techniques (e.g. [GHP13, KZ16]), our techniques do not require increased numbers of queries.

6.1 Overview of amplification

Existing compression side-channel attacks are known to be sensitive to noise. These attacks rely on length differences in the compressed data of no more than a few bytes, whereas a large part of the plaintext to be compressed may be unknown to the attacker or change on each query, and the target may also deliberately introduce noise to the compression process as a heuristic defence against compression side-channel attacks (Section 2.3). Even when the aforementioned noise is absent, the noise from Huffman coding in DEFLATE may already become a hurdle to the attacker. Statistical methods can mitigate noise, but at the cost of performing more queries, making the attack less practical and easier to detect.

As a concrete example, we consider a simple compression side-channel attack that checks whether $ck[1] = g$ for some byte g. In order to do so, the attacker may make a query $Q = P \| g$. If the first byte in ck is indeed g, then LZ77 is expected to encode the query as $\texttt{<L}(\ell_P + 1)\texttt{,D*>}$, a back-reference to its previous occurrence; otherwise, LZ77 is expected to encode the query as $\texttt{<L}\ell_P\texttt{,D*>} \| g$, only replacing P in the query with a back-reference while keeping the byte g as is. Therefore, the lengths of the LZ77 outputs differ by one symbol depending on whether $ck[1] = g$ or not, and the length difference likely persists in the compressed data after Huffman coding, from which the attacker can obtain information about whether $ck[1] = g$.

In the above example, the compressed lengths when $ck[1] = g$ and when $ck[1] \neq g$ can hardly differ by more than one or two bytes, because (i) the LZ77 output is virtually the same in the two cases, and (ii) the Huffman codes are mostly determined by other parts of the plaintext. However, the contrapositive of the above arguments suggests that, in order to increase the compressed length difference, one might first try to design queries for which either (i) or (ii) does not hold. In particular, because the behaviour of the LZ77 algorithm depends on the input it has seen, it appears possible to design queries for which the LZ77 outputs can be very different with different secrets, which may be used to amplify the compressed length difference, making the side-channel signal more visible to the attacker. As we will see in this section, the adaptivity of LZ77 enables several different ways to design amplification queries, and we will also describe how to exploit the hash table implementation of LZ77 for amplification in Section 7.

We describe our amplification techniques in the setting where an attacker wants to check whether $ck[1]$ is equal to a guess $g \in \Omega$, which, as we have described in Section 5.1, can be extended to a secret recovery attack. We will further generalise our amplification techniques to *amplification gadgets* in Section 8.

Query structure. The queries we design follow the general structure we describe below, which we will also use in Sections 7 and 8.

A query Q may contain up to four parts, each represented as a byte string in Σ^*. Generally,

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

where dict_1 and dict_2 are two *dictionary strings*, filler is an optional *filler string* between the two dictionary strings, and $\mathsf{gadgets}$ is a string of gadgets. It is not necessary for dict_1, filler, dict_2 and $\mathsf{gadgets}$ to be next to each other, as long as they follow this order in the plaintext. In practice, 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 sequence of strings called a *dictionary*. In some cases, dict_1 and dict_2 can also be omitted. Duplicated strings in a dictionary may be removed for all queries constructed using our techniques in Sections 6 to 8. One way to construct a dictionary string dict from a sequence of strings $(S_1, \ldots, S_d)$ is

$$\mathsf{dict} = \mathsf{sep}_{\mathsf{dict}} \| S_1 \| \mathsf{sep}_{\mathsf{dict}} \| S_2 \| \mathsf{sep}_{\mathsf{dict}} \| \cdots \| \mathsf{sep}_{\mathsf{dict}} \| S_d \| \mathsf{sep}_{\mathsf{dict}},$$

where the separator $\mathsf{sep}_{\mathsf{dict}}$ is a very short string (e.g. of one byte) that preferably never or only rarely appears in other parts of the plaintext. The dictionary strings dict_1 and dict_2 can share the same separator $\mathsf{sep}_{\mathsf{dict}}$. For example, a dictionary string for the sequence $(\texttt{foo}, \texttt{bar})$ can be `$foo$bar$`, where we use `$` as the separator. Some of our techniques may require different constructions of dictionary strings when applied to certain compressor configurations, which we will discuss separately later.

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 byte repeated multiple times or a random byte 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 gadget in Section 8. For ease of understanding, we do not present our amplification techniques from the perspective of gadgets in this section, but we note that our telescoping technique (Section 6.2) constructs $\mathsf{gadgets}$ as a telescoped-match gadget (Section 8.3.3), and our chaining techniques (Section 6.3) construct $\mathsf{gadgets}$ as the concatenation of a match gadget (Section 8.3.1) and an amplify gadget (Section 8.4).

Formal requirements. Note that we give formal requirements for our techniques in this section as well as in Sections 7 and 8 to enable rigorous analyses; under suitable parameters, most of these requirements hold with high probabilities in practice, and some can even be weakened or dropped without affecting the correctness of our techniques.[60] For simplicity, we generally do not specify the position of the sliding window when we state the requirements for our techniques.[61]

6.2 Telescoping

Telescoping is a simple but convenient technique for amplification. It builds a query that repeats a guess multiple times with successively longer (or shorter) prefixes. Telescoping can be used to achieve a compressed length difference roughly up to the prefix length ℓ_P using relatively short queries, making it particularly useful in cases where the maximum query length L_Q is small. It is also possible to perform telescoping in concert with other amplification techniques (Section 8.3).

Akagi et al. have used similar techniques to derive lower bounds on the sensitivities of several LZ-family compressors [AFI23], and Inenaga provided an example that demonstrates their techniques on the

[60] Notably, our implementation only enforces a subset of the requirements; see Section 9.1.

[61] It usually suffices to consider the sliding window when the LZ77 pointer is just before the gadget string $\mathsf{gadgets}$. However, for a precise interpretation, one should refer to the analysis and check the sliding windows for all LZ77 pointer positions where the requirement is invoked with regard to the strings in question.

7-Zip compression software.[62] However, to our knowledge, no previous work used telescoping or similar techniques in the context of compression side-channel attacks.

Intuitively, the telescoping technique can be seen as a straightforward improvement to a naive attempt at amplifying the compression side channel, in which the attacker simply repeats the known prefix and the guess $P\|g$ for $m > 1$ times in the query, possibly with separators between repetitions, hoping that LZ77 would output $\texttt{<L}(\ell_P + 1)\texttt{,D*>}$ for each repetition if $ck[1] = g$, and $\texttt{<L}\ell_P\texttt{,D*>}\|g$ otherwise, resulting in a compressed length difference proportional to m.

The issue with the naive attempt is that, for all but the first occurrence of $P\|g$ in the query, any match starting at a previous occurrence of $P\|g$ is closer and at least as long as the match starting at $P\|ck$ (the actual prefix and cookie located before the query), and is therefore more favourable. As a result, LZ77 would encode most of the query independently from the value of $ck[1]$, and the compressed length difference would be small and not scale with m. To circumvent this issue, the attacker can modify the query to use a successively longer suffix of P for each repetition. Then, for each repetition, the match at $P\|ck[1]$ is longer than any possible match within the query and thus more favourable.

Construction. Telescoping normally does not require dict_1, filler or dict_2 and only uses $\mathsf{gadgets}$. However, the attacker should add the prefix P to dict_1 or dict_2 to exclude unwanted matches in attack scenarios where P may not always appear in the sliding window.[63] Suppose the attacker wishes to check whether the byte $ck[1]$ is equal to a guess $g \in \Omega$ using telescoping. The attacker chooses $m \in \mathbb{N}_+$ where $m \leq \ell_P + 2 - \mathsf{min_match}$, and queries

$$Q = P\,[m : \ell_P]\,\|g\|\mathsf{sep}_1\|P\,[m-1 : \ell_P]\,\|g\|\mathsf{sep}_2\|\cdots\|\mathsf{sep}_{m-1}\|P\,[1 : \ell_P]\,\|g,$$

where the separators $\mathsf{sep}_1, \ldots, \mathsf{sep}_{m-1}$ between the segments are pairwise distinct byte strings such that their first bytes $\mathsf{sep}_1[1], \ldots, \mathsf{sep}_{m-1}[1]$ are preferably not in Ω. It usually suffices to use separators of one or two bytes each, but longer separators also work.[64] The separators $\mathsf{sep}_1, \ldots, \mathsf{sep}_{m-1}$ can be omitted if $P[i] \notin \Omega$ and $P[i] \neq P[i+1]$ hold for every $i \in [m-1]$; the intuition is that $P[m-1], \ldots, P[1]$, the first bytes in all but the first segment, can already serve the functionality of separators.

In our example setting, for $g = \texttt{1}$ and $m = \ell_P - 2 = 5$, the query Q can be

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

with separators `A`, `B`, `C`, `D`, or simply

```
et=1ret=1cret=1ecret=1secret=1
```

as no adjacent bytes in the prefix `secret=` are the same and no digits appear in the prefix. Note that, if $m = 1$, then the query Q degenerates to $P\|g$, or to `secret=1` in our example.

If $g = ck[1]$, then LZ77 likely encodes the query Q as

$$\texttt{<L}(\ell_P - m + 2)\texttt{,D*>}\|\mathsf{sep}_1\|\texttt{<L}(\ell_P - m + 3)\texttt{,D*>}\|\mathsf{sep}_2\|\cdots\|\mathsf{sep}_{m-1}\|\texttt{<L}(\ell_P + 1)\texttt{,D*>}.$$

Otherwise, if $g \neq ck[1]$, then LZ77 likely encodes Q as

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

where LZ77 likely outputs $P\,[m : \ell_P]$ instead of $\texttt{<L}(\ell_P - m + 1)\texttt{,D*>}$ for the first part of the query if $m = \ell_P + 2 - \mathsf{min_match}$ (because $P\,[m : \ell_P]$ would be too short), and for $i \in [m-1]$, the encoding of the separator sep_i might be different than stated if $|\mathsf{sep}_i| \geq \mathsf{min_match}$, but its encoding is expected to be independent of whether $g = ck[1]$ or not.

Requirements. In addition to the aforementioned requirements, we require that the following conditions hold.

[62] See slide 29, https://shunsuke-inenaga.github.io/slides/Sequences_in_London_2024_inenaga.pdf, or slide 32 (in Japanese), https://shunsuke-inenaga.github.io/slides/LA2021_winter_inenaga.pdf.

[63] We will discuss this further in Section 8.3.

[64] In certain attack scenarios outside of our model, the attacker may only be able to control multiple short parts of the plaintext (e.g. usernames in [PST23]), in which case the attacker can place individual segments $P[i, \ell_P]\|g$ in different parts of the plaintext, and the plaintexts between these segments can be seen as the separators.

1. For each $i \in [m]$, LZ77 can find the $P[i:\ell_P]\|ck[1]$ in the sliding window just before $P[i:\ell_P]\|g$ in the query Q, but $P[i:\ell_P]$ and $P[i+1:\ell_P]\|g$ do not appear elsewhere in the same sliding window besides (possibly) appearing in $P\|ck$.
2. For each $i \in [m-1]$, let $\ell_{s,i} = |\mathsf{sep}_i|$; if $\ell_{s,i} > 0$, then $g\|\mathsf{sep}_i[1:\min\{\ell_{s,i}, \mathsf{min_match}-1\}]$, a substring of $g\|\mathsf{sep}_i$ in Q, does not appear in the sliding window, and for $j \in [\ell_{s,i}]$ where $j > \ell_{s,i} - \mathsf{min_match} + 1$, the byte string $\mathsf{sep}_i[j:\ell_{s,i}]\|P[m-i+1:m-i-\ell_{s,i}+j+\mathsf{min_match}-1]$ of length $\mathsf{min_match}$, a substring of $\mathsf{sep}_i\|P[m-1:\ell_P]$ in Q, does not appear in the sliding window.

Analysis. We now justify the output of LZ77 by discussing the case where lazy matching is enabled, $m < \ell_P + 2 - \mathsf{min_match}$, and $0 < |\mathsf{sep}_i| < \mathsf{min_match}$ for each $i \in [m-1]$; other cases follow similarly. We assume that the LZ77 pointer is initially at the first byte of the gadget string $\mathsf{gadgets}$.[65]

- If $g = ck[1]$, then by condition 1, LZ77 likely finds a match for $P[m:\ell_P]\|g$ of length $\ell_P - m + 2 \geq \mathsf{min_match}$ within $P\|ck[1]$ in the sliding window, which by condition 1 is unlikely to be improved by lazy matching. Therefore, LZ77 likely outputs a back-reference `<L(`ℓ_P-m+2`),D*>` and moves the pointer to the first byte of sep_1. By condition 2 and our assumption on $|\mathsf{sep}_1|$, LZ77 does not find any match starting within sep_1, outputs sep_1 verbatim, and moves the pointer to the start of $P[m-1:\ell_P]\|g$. The rest follows in the same manner.
- If $g \neq ck[1]$, then by condition 1 and our assumption on m, LZ77 likely finds a match for $P[m:\ell_P]$ of length $\ell_P - m + 1 \geq \mathsf{min_match}$ within the known prefix P in the sliding window, which by condition 1 is unlikely to be improved by lazy matching. Therefore, LZ77 likely outputs a back-reference `<L(`ℓ_P-m+1`),D*>` and moves the pointer to the guess g immediately after $P[m:\ell_P]$. By condition 2 and our assumption on $|\mathsf{sep}_1|$, LZ77 does not find any match starting within $g\|\mathsf{sep}_1$, outputs $g\|\mathsf{sep}_1$ verbatim, and moves the pointer to the start of $P[m-1:\ell_P]\|g$. The rest follows in the same manner.

Performance. The lengths of the LZ77 output likely differ by m symbols depending on whether $g = ck[1]$ or not, so after Huffman coding, the compressed length when $g = ck[1]$ is likely smaller than that when $g \neq ck[1]$ by around m bytes. However, the exact length difference depends on Huffman coding: for example, if m is large, then the literal byte g may be encoded more efficiently by Huffman coding, making the length difference smaller. The length of the query is $\ell_Q = m(\ell_P - m + 2)(\ell_P + 1)/2$ if we omit the separators. Because $m \leq \ell_P + 2 - \mathsf{min_match}$, the compressed length difference is roughly upper bounded by ℓ_P, which means telescoping can only increase the compressed length difference by a limited amount if the known prefix is short.

Reverse telescoping. We note that it is possible to reverse the direction of telescoping in some cases, where the attacker can build a perhaps counter-intuitive telescoping query with successively shorter prefixes. If lazy matching is disabled, $m < \ell_P - \mathsf{max_insert} + 2$, and $P[i:\ell_P] \neq P[i+1:\ell_P]\|g$ for each $i \in [m-1]$, then the attacker may query

$$Q = P[1:\ell_P]\|g\|\mathsf{sep}_1\|P[2:\ell_P]\|g\|\mathsf{sep}_2\|\cdots\|\mathsf{sep}_{m-1}\|P[m:\ell_P]\|g,$$

or

`secret=1ecret=1cret=1ret=1`

in our example setting for $g =$ `1` and $m = 4$, with the separators omitted. The effect of the query is similar to that of the normal telescoping construction. For $i \in [m-1]$, it may appear that LZ77 would find a match for the byte string $P[i+1:\ell_P]\|g$ in the query within its preceding segment, $P[i:\ell_P]\|g$, irrespective of whether $ck[1] = g$ or not; however, because $\ell_P - i + 1 \geq \ell_P - m + 2 > \mathsf{max_insert}$, LZ77 in fact only inserts the substring starting at the first byte of $P[i:\ell_P]\|g$ into the hash table, and cannot find any back-reference that points to its second byte, so the LZ77 encoding of $P[i+1:\ell_P]\|g$ does depend on whether $ck[1] = g$. In our example setting, LZ77 likely first outputs either `<L8,D*>` or `<L7,D*>1` depending on whether $ck[1] =$ `1` or not, and moves the pointer to the first byte of the second segment, `ecret=1`. There, LZ77 cannot find a back-reference `<L7,D7>` to the first segment, as the substring `ecr` in `secret=1` before the query was not inserted into the hash table; instead, LZ77 likely outputs either `<L7,D*>` or `<L6,D*>1` depending on whether $ck[1] =$ `1` or not, and moves the pointer to the next segment. The rest follows in the same manner. However, there is no apparent reason to prefer reversing the direction of telescoping.

[65] We will see in Section 8 that this can be enforced by prepending an align gadget (Section 8.2.1) to $\mathsf{gadgets}$.

6.3 Chaining

As we have seen in Section 6.2, a main disadvantage of the telescoping technique is that the compressed length difference resulting from telescoping alone is limited to roughly ℓ_P bytes, which will be hard to observe if a large amount of noise is present. In this section, we describe several techniques for amplification that we refer to collectively as *chaining*, which can produce larger compressed length differences at the cost of using longer queries.

We motivate the intuition behind the chaining techniques by making the following observation. When the attacker queries $P\|g$, depending on whether $ck[1] = g$, LZ77 not only encodes the query differently, but also moves its pointer to different positions in the plaintext: if $ck[1] = g$, then LZ77 replaces the whole query with a single back-reference `<L`(ℓ_P+1)`,D*>` and moves the pointer to the byte immediately after the query; otherwise, LZ77 only replaces P in the query with a back-reference `<L`ℓ_P`,D*>` 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 byte in the next step and coincides with the pointer position when $ck[1] = g$. However, suppose one can construct a set of strings that LZ77 encodes differently depending on the initial pointer position, then it is possible to create compressed length differences from pointer position differences by attaching one of these strings to $P\|g$ in the query. Moreover, if the strings in question also maintain the difference in pointer positions, then one can potentially chain many of these strings together, so that LZ77 produces substantially different output depending on whether $ck[1] = g$.

In this section, we describe two chaining techniques, *distance-based chaining* and *match-based chaining*. The collision-based amplification technique in Section 7 can also be seen as a type of chaining.

Query structure. The chaining techniques we describe in this section share the following structure.

A chaining query consists of four parts, dict_1, filler, dict_2 and $\mathsf{gadgets}$, as specified in Section 6.1. The content of the dictionary strings dict_1 and dict_2 differs for different techniques and will be specified later. The optional filler string filler is constructed as described in Section 6.1. The string of gadgets $\mathsf{gadgets}$ is of the form $P\|g\|S$, where the byte string S represents an *amplification chain*, to be defined below. The query is of the form

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

We require that the substring $P\|ck[1]$ can be found by LZ77 in the sliding window, but $P[2:\ell_P]\,\|g\|S[1]$ does not not appear in the sliding window. Therefore, when the LZ77 pointer is at $P[1]$ in the query, if $ck[1] = g$, then LZ77 finds a match for $P\|g$ in the sliding window, and moves the pointer to $S[1]$ in the query; otherwise, if $ck[1] \neq g$, then LZ77 finds a match for P and moves the pointer to the position of the guess g in the query.[66]

The amplification chain S is a byte string of length $\ell_S \in \mathbb{N}_+$ chosen by the attacker, which we will generalise to an amplification gadget in Section 8.4. The chain S consists of $m \in \mathbb{N}_+$ byte strings that we refer to as *segments*, where $m \geq 2$, and most or all of the segments are of length $k \in \mathbb{N}_+$, where $k \geq \mathsf{min_match}$. By definition, the chain length ℓ_S is approximately mk, and we will express ℓ_S as a function of m and k for each technique. A larger number of segments m generally increases the compressed length difference, but it also increases the query size and the probability of failure; a higher segment length k generally decreases the probability of failure and makes the chain more likely to work as desired, but it also increases the query size and may make the query less space-efficient.[67] For the techniques we introduce in this section, the length of the query ℓ_Q is bounded by the maximum query length L_Q as well as the sliding window size $\mathsf{window_size}$.[68]

The chain S should not contain special bytes that we designate for other purposes, such as the separators for dict_1 and dict_2, and the first byte $S[1]$ preferably should not take value in Ω. While it suffices for the rest of the chain $S[2:\ell_S]$ to be a random byte string of length $\ell_S - 1$, we will also give other constructions that can yield smaller probabilities of failure or larger compressed length differences.

If the attacker knows some parts of the plaintext located before the query, then they may use these known byte strings to construct the query, reducing the size of dict_1.

[66] Here we implicitly constructed a match gadget $P\|g$, stated its requirements and gave an analysis; see Section 8.3.1 for a formal description of the match gadget.

[67] Intuitively, each of the m segments require the presence or absence of certain strings of length roughly the segment length k in the sliding window to work correctly, so a larger m and a smaller k increases the probability that some requirements are not satisfied.

[68] More precisely, the query may also be slightly larger than $\mathsf{window_size}$ as long as its requirements are satisfied.

To simplify the description and analysis of chaining, we abuse notation to let $S[0]$ denote the byte immediately before S in the plaintext (i.e. g), and for each $i \in \mathbb{N}_+$, we let $S[\ell_S + i]$ denote the i-th byte located after S in the plaintext, or a special end symbol if there are fewer than i bytes after S.

6.3.1 Distance-based chaining

Distance-based chaining constructs a *distance chain* that amplifies the compressed length difference by exploiting the combination of LZ77 and Huffman coding in DEFLATE. More specifically, depending on whether the guess g is equal to $ck[1]$ or not, LZ77 encodes the distance chain S in the query to two different sequences of back-references and literals, such that the back-references tend to be of shorter distances and are therefore encoded more efficiently by Huffman coding when $g = ck[1]$, or vice versa. The distance-based chaining technique shows the importance of studying DEFLATE as a whole, as this technique would not work if only LZ77 or Huffman coding were used for compression.

Recall from Section 3.3.2 that, to encode the distance of a back-reference `<L`x`,D`y`>`, DEFLATE first maps the distance y to a distance symbol between 0 and 29 and some extra bits and then encodes the distance symbol using Huffman coding. The extra bits do not participate in Huffman coding and remain unchanged in the compressed output. The number of extra bits is determined by the distance symbol, and it increases as the distance 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 1). As a result, longer distances tend to require more space to encode.

Construction. A distance chain S is a byte string of length $\ell_S = mk$. The dictionary string dict_1 represents the sequence of byte strings $(S[(i-1)k : ik-1])_{i\in[m]}$, and dict_2 represents the sequence of byte strings $(S[(i-1)k+1 : ik])_{i\in[m]}$.

In our example setting, for $g = 1$, $m = 5$ and $k = 3$, the query Q can be

```
-1AB-CDE-FGH-IJK-LMN-------ABC-DEF-GHI-JKL-MNO-secret=1ABCDEFGHIJKLMNO
```

where we use `-` as separators in dict_1 (`-1AB-CDE-FGH-IJK-LMN-`) and dict_2 (`-ABC-DEF-GHI-JKL-MNO-`), and `-----` as the filler string filler. The gadget string $\mathsf{gadgets}$ is `secret=1ABCDEFGHIJKLMNO`.

If $ck[1] = g$, then LZ77 encodes $\mathsf{gadgets} = P\|g\|S$ in the query as

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

where the back-references correspond to

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

which are `secret=1`, `ABC`, `DEF`, `GHI`, `JKL`, `MNO` in our example. All back-references except the first one are expected to point to a byte string in dict_2.

Otherwise, if $ck[1] \neq g$, then LZ77 encodes $\mathsf{gadgets} = P\|g\|S$ in the query as

$$\texttt{<L}\ell_P\texttt{,D*>}\|\texttt{<L}k\texttt{,D*>}\|\cdots\|\texttt{<L}k\texttt{,D*>}\|S[mk],$$

where the back-references and literals correspond to

$$P, S[0:k-1], \ldots, S[(m-1)k : mk-1], S[mk],$$

which are `secret=`, `1AB`, `CDE`, `FGH`, `IJK`, `LMN`, `O` in our example. All back-references except the first one are expected to point to a byte string in dict_1.

The lengths of LZ77 output differ by only one symbol depending on whether $ck[1] = g$ or not. However, if $ck[1] = g$, then the back-reference distances are generally smaller and therefore need fewer extra bits to encode; this is because most back-references refer to strings in the closer dictionary dict_2, as opposed to referring to strings in the farther dictionary dict_1 if $ck[1] \neq g$. Since the extra bits stay unchanged during Huffman coding, if we assume Huffman coding encodes other parts of LZ77 output to similar lengths, then differences in the number of extra bits will be reflected the difference of compressed lengths, making the compressed length smaller when $ck[1] = g$.

We remark that, by swapping the content of dict_1 and dict_2 in the query, one can immediately obtain a distance chain that amplifies compressed length differences in the reverse direction, i.e. it increases the compressed length when $ck[1] = g$.[69]

[69] We will see in Section 8 that one can reverse the direction of amplification for any amplification gadget by prepending and appending a NOT gadget to the amplification gadget in question, but the generic construction is more cumbersome.

Requirements. For distance-based chaining to work correctly with high probability, we require 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, 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, 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 constructs the distance chain S by sampling each byte independently and uniformly at random from a subset of Σ of reasonable size, but many other constructions are also feasible, including the optimisations that we will introduce later.

Analysis. We now justify the output of LZ77 for $\mathsf{gadgets} = P\|g\|S$. We assume the LZ77 pointer is initially at the first byte of P.

- If $ck[1] = g$, then LZ77 finds a match for $P\|g$ in the sliding window and moves the pointer to $S[1]$ in the query Q. Next, by condition 2, LZ77 finds a match for $S[1:k]$ in the sliding window, likely in dict_2, which by condition 2 cannot be extended or improved by lazy matching, and moves the pointer to $S[k+1]$ in the query. Continuing this process, we conclude that LZ77 outputs back-references for $S[1:k], S[k+1:2k], \ldots, S[(m-1)k+1:mk]$, likely all in dict_2, and moves the pointer to to $S[mk+1]$.
- Otherwise, if $ck[1] \neq g$, then LZ77 finds a match for P and moves the pointer to g in the query Q. Next, by condition 1, LZ77 finds a match for $S[0:k-1] = g\|S[1:k-1]$ in the sliding window, likely in dict_1, which by condition 1 cannot be extended and by condition 2 cannot be improved by lazy matching, and moves the pointer to $S[k]$ in the query. Continuing this process, we conclude that LZ77 outputs back-references for $S[0:k-1], S[k:2k-1] \ldots, S[(m-1)k:mk-1]$, likely all in dict_1, and moves the pointer to $S[mk]$.

Performance. The Huffman coding process is generally stable and only affects amplification minimally. It appears possible to construct adversarial examples in which Huffman coding partially or completely offsets the length change from extra bits, but such examples are unlikely to occur in practice.

It is challenging to deduce the expected compressed length difference for distance-based chaining; 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 a distance chain creates a compressed length difference of around $m/4$ bytes. The exact query length depends on the constructions of dict_1, dict_2 and filler; if the dictionaries use one-byte separators and the filler is empty, then the query length ℓ_Q is around $(3k+2)m$ bytes. Because the number of extra bits grows much slower than the distance, the actual compressed length difference may be much smaller than estimated when m is large.[70] Distance-based chaining can create larger compressed length differences than telescoping, but telescoping is more effective for amplification with very short queries.

We now briefly discuss two ways to reduce query sizes and increase compressed length differences for the distance-based chaining technique by using distance chains with special structures.

Optimisation via doubling. A natural idea is to construct the chain S in a specific way such that dict_1 or dict_2 contains duplicated entries that can be removed, so that the dictionary string in question represents fewer than m byte strings and is thus shorter than what is required by the basic construction. One strategy to reduce the entries in dictionary strings is *doubling*, which makes every second segment in S the same as its preceding segment. Doubling not only reduces the size of dict_2 by half but also exploits Huffman coding to increase the compressed length difference.

To simplify our description, we assume $m = 2m' - 1$ for some $m' \in \mathbb{N}_+$ such that $m' \geq 2$. Our construction requires $k > 2$, which is usually satisfied trivially because $k \geq \mathsf{min_match}$. Let S' be a byte string of length $m'k$. We construct the distance chain S of length $\ell_S = mk$ as

$$S'[1:k] \,\|\, S'[k+1:2k] \,\|\, S'[k+1:2k] \,\|\, \cdots \|\, S'[(m'-1)k+1:m'k] \,\|\, S'[(m'-1)k+1:m'k].$$

The dictionary string dict_1 represents the sequence of byte strings $(S[(i-1)k : ik-1])_{i\in[m]}$, which is

$$(g\|S'[1:k-1], S'[k:2k-1], S'[2k]\|S'[k+1:2k-1], \ldots, S'[m'k]\|S'[(m'-1)k+1:m'k-1]).$$

[70] See Fig. 11a, Section 9.3 for experimental results.

The dictionary string dict_2 represents the sequence of byte strings $(S[(i-1)k+1:ik])_{i\in[m]}$, which is

$$(S'[1:k], S'[k+1:2k], S'[k+1:2k], \ldots, S'[(m'-1)k+1:m'k], S'[(m'-1)k+1:m'k]).$$

By removing the at least $(m-1)/2$ repeated entries in the sequence, one can construct dict_2 using around $(m+1)(k+1)/2$ bytes, roughly halving the number of bytes normally required. We do not repeat the first segment of the chain, $S'[1:k]$, in order to lower its probability of failure.

In our example setting, for $g = \mathtt{1}$, $m = 5$ and $k = 3$, a distance-based chaining query constructed using doubling can be

```
-1AB-CDE-FDE-FGH-IGH-------ABC-DEF-GHI-secret=1ABCDEFDEFGHIGHI
```

for $S' = \mathtt{ABCDEFGHI}$.

A key benefit of doubling is that it enlarges the differences in the LZ77 back-reference distances. If $ck[1] = g$, then LZ77 likely encodes $\mathsf{gadgets} = P\|g\|S$ in the query as

$$\mathtt{<L}(\ell_P+1)\mathtt{,D*>}\|\mathtt{<L}k\mathtt{,D*>}\|\mathtt{<L}k\mathtt{,D*>}\|\mathtt{<L}k\mathtt{,D}k\mathtt{>}\|\cdots\|\mathtt{<L}k\mathtt{,D*>}\|\mathtt{<L}k\mathtt{,D}k\mathtt{>},$$

where about half of the back-references point to its previous segment in the chain rather than dict_2. The corresponding distances require even fewer extra bits to encode as k is small. Otherwise, if $ck[1] \neq g$, then almost all of the back-references still point to dict_1; while dict_1 is now closer to the chain, because the distance between dict_1 and the chain is large, the reduction in the number of extra bits is insignificant compared to the case where $ck[1] = g$. Therefore, the compressed length difference is expected to increase with doubling, and the effect is even more significant when m is large.

For the reverse direction of amplification, we can perform doubling using a similar construction.[71]

We note that the use of structured chains in doubling increases its probability of failure if we simply sample S' uniformly at random from $\Sigma^{(m+1)k/2}$, in which case it is less likely for the stipulated conditions for distance-based chaining to hold. One mitigation that works for both normal and reverse direction of amplification is to select S' such that for each $i_1 \in [m'-1]$, $S'[i_1k] \neq S'[(i_1+1)k]$, and for each $i_2 \in [m'-2]$, $S'[i_2k+1] \neq S'[(i_2+1)k+1]$.

Optimisation via overlapping. Another idea for reducing the query size is to construct S such that dict_1 or dict_2 can be encoded more efficiently. One such strategy is *overlapping*, in which we construct S carefully such that we can encode one of the dictionaries as a single long byte string that contains every entry of the dictionary as a substring, and the entries overlap each other in the long byte string. Compared to doubling, overlapping is generally less effective and more prone to errors.

We now formally describe the overlapping construction. Let $\lambda \in [k-1]$, and let S' be a byte string of length $(m-1)\lambda + k$. We construct the distance chain S of length $\ell_Q = mk$ as

$$S'[(m-1)\lambda+1:(m-1)\lambda+k]\,\|S'[(m-2)\lambda+1:(m-2)\lambda+k]\,\|\cdots\|S'[\lambda+1:\lambda+k]\,\|S'[1:k].$$

The dictionary string dict_1 represents the sequence of byte strings $(S[(i-1)k:ik-1])_{i\in[m]}$, which is

$$(g\|S'[(m-1)\lambda+1:(m-1)\lambda+k-1], S'[(m-1)\lambda+k]\|S'[(m-2)\lambda+1:(m-2)\lambda+k-1],$$
$$\ldots, S'[\lambda+k]\|S'[1:k-1]).$$

The dictionary string dict_2 is simply S', which contains every byte string in $(S[(i-1)k+1:ik])_{i\in[m]}$ as a substring. Compared to the unoptimised construction, dict_2 is shorter by around $m(k-\lambda)$ bytes. As λ decreases, dict_2 becomes shorter, but at the same time the query becomes less likely to work as desired.

In our example setting, for $g = \mathtt{1}$, $m = 5$, $k = 3$ and $\lambda = 1$, a distance-based chaining query constructed using overlapping can be

```
-1EF-GDE-FCD-EBC-DAB------ABCDEFG-secret=1EFGDEFCDEBCDABC
```

for $S' = \mathtt{ABCDEFG}$.

[71] Simply swapping dict_1 and dict_2 may not work as desired, since if $ck[1] = g$, then about one half of the back-references point to the new dict_1, and require more extra bits to encode their distances, but another half of the back-references have a distance of k and require fewer extra bits to encode, partially or completely offsetting the compressed length difference; the construction of the distance chain S must also be modified.

If $\mathsf{max_insert}$ is defined, then we require that $k \leq \mathsf{max_insert}+1$; otherwise, LZ77 can find matches in dict_1 longer than $\mathsf{max_insert}$ for some substrings in dict_2, in which case, by the definition of $\mathsf{max_insert}$, some other substrings in dict_2 will not be added to the LZ77 hash table, potentially affecting the correctness of the query.

To reverse the direction of amplification, we can simply swap the content of dict_1 and dict_2.

Because the distance chain S is structured and we use a non-canonical construction for one dictionary string, care should be taken to ensure that the query still works with high probability. For example, when $\lambda = 1$, then one should select S' such that $S'[i] \neq S'[i+k-1]$ for $1 \leq i \leq m-1$.

6.3.2 Match-based chaining

Match-based chaining constructs a *match chain* that amplifies the compressed length difference by exploiting the greedy strategy of LZ77 in DEFLATE. We carefully design the query such that, depending on whether the guess g is equal to $ck[1]$ or not, LZ77 encodes the match chain S in the query to two different sequences of back-references and literals, where the sequence for the case $g = ck[1]$ is markedly shorter and results in shorter compressed plaintext after Huffman coding than the case $g \neq ck[1]$, or vice versa. This is in contrast to distance-based chaining, which does not directly produce visible differences in LZ77 output lengths, but rather exploits the combination of LZ77 and Huffman coding.

Match-based chaining is generally preferable to distance-based chaining, because match-based chaining typically achieves larger compressed length differences than distance-based chaining with queries of similar lengths. However, it should be taken into account that match chains have more complex structures and are therefore more likely to fail.[72]

Our constructions of match-based chaining queries follow the general query structure for chaining described in this subsection; i.e.

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

Our constructions do not require the filler string filler to work, but similar to distance-based chaining, using a filler string can enlarge the differences in distances for back-references pointing to dict_1 and those pointing to dict_2, thereby increasing the compressed length difference by a small amount.

We first describe how to construct a match chain when lazy matching is disabled or $\mathsf{max_lazy}$ is small. Then, we discuss the more interesting and simultaneously more challenging case where lazy matching is enabled.

Fast match chain: construction. We first describe the construction of a *fast match chain*, designed primarily for exploiting the $\mathsf{deflate_fast}$ compression strategy. Note that a fast match chain becomes inefficient and should not be used when lazy matching is enabled, unless $\mathsf{max_lazy}$ is small.

Let $k' \in [k-1]$, such that $k' \geq \max\{\mathsf{min_match}, k - \mathsf{min_match}+2\}$, and, if lazy matching is enabled, $k' \geq \mathsf{max_lazy}$. A fast match chain S is a byte string of length $\ell_S = mk$. The dictionary string dict_1 represents the sequence of m byte strings

$$\left(S\left[0 : k'-1\right], S\left[k : k+k'-1\right], \ldots, S\left[(m-2)k : (m-2)k+k'-1\right], S\left[(m-1)k : mk-1\right]\right)$$

of total length $(m-1)k'+k$, and dict_2 represents the sequence of m byte strings $\{S\left[(i-1)k+1 : ik\right]]\}_{i\in[m]}$ of total length mk.

In our example setting, for $g = \texttt{1}$, $m = 5$, $k = 4$ and $k' = 3$, the query can be

```
-1AB-DEF-HIJ-LMN-PQRS--ABCD-EFGH-IJKL-MNOP-QRST-secret=1ABCDEFGHIJKLMNOPQRST
```

where we use - as separators in dict_1 and dict_2, and we set the filler string filler to the empty string ε.

If $ck[1] = g$, then LZ77 encodes $\mathsf{gadgets} = P\|g\|S$ in the query as

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

where the back-references correspond to

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

which are `secret=1`, `ABCD`, `EFGH`, `IJKL`, `MNOP`, `QRST` in our example. All back-references except the first one are expected to point to a byte string in dict_2.

[72] See Fig. 11a, Section 9.3 for experimental results.

Otherwise, if $ck[1] \neq g$, then LZ77 encodes $\mathsf{gadgets} = P\|g\|S$ in the query as a mixture of back-references and literals:

$$\texttt{<L}\ell_P\texttt{,D*>}\|\texttt{<L}k'\texttt{,D*>}\|S\left[k' : k-1\right]\|\cdots\|\texttt{<L}k'\texttt{,D*>}\|S\left[(m-2)k+k' : (m-1)k-1\right]\|\texttt{<L}k\texttt{,D*>}\|S[mk],$$

where the back-references and literals correspond to

$$P, S\left[0 : k'-1\right], S\left[k' : k-1\right], \ldots, S\left[(m-2)k : (m-2)k+k'-1\right],$$
$$S\left[(m-2)k+k' : (m-1)k-1\right], S\left[(m-1)k : mk-1\right], S[mk],$$

which are `secret=`, `1AB`, `C`, `DEF`, `G`, `HIJ`, `K`, `LMN`, `O`, `PQRS`, `T` in our example. All back-references except the first one are expected to point to a byte string in dict_1.

Fast match chain: requirements. For the fast match chain to work correctly, we require that:

1. For each $i \in [m-1]$, $S\left[(i-1)k : (i-1)k+k'-1\right]$ in dict_1 can be found by LZ77 in the sliding window, but $S\left[(i-1)k : (i-1)k+k'\right]$ does not appear in the sliding window.
2. The byte string $S\left[(m-1)k : mk-1\right]$ in dict_1 can be found by LZ77 in the sliding window, but $S\left[(m-1)k : mk\right]$ does not appear in the sliding window.
3. For each $i \in [m-1]$ and $j \in \mathbb{N}_+$ such that $(i-1)k+k' \leq j \leq ik-1$, $S\left[j : j+\mathsf{min_match}-1\right]$ does not appear in the sliding window.
4. For each $i \in [m]$, $S\left[(i-1)k+1 : ik\right]$ in dict_2 can be found by LZ77 in the sliding window, but $S\left[(i-1)k+1 : ik+1\right]$ does not appear in the sliding window.

Under reasonable parameters, the above conditions likely hold if one constructs the fast match chain S by sampling each byte independently and uniformly at random from a large subset of Σ, but many other constructions are also feasible, such as the doubling optimisation we will describe below.

Fast match chain: analysis. We now justify the output of LZ77 for $\mathsf{gadgets} = P\|g\|S$. We assume the LZ77 pointer is initially at the first byte of P.

- If $ck[1] = g$, then LZ77 finds a match for $P\|g$ in the sliding window and moves the pointer to $S[1]$ in the query Q. Next, by condition 4, LZ77 finds a match for $S\left[1 : k\right]$ in the sliding window, likely in dict_2. By condition 4, the match cannot be extended, and lazy matching is either disabled or not performed because the match is of length $k > k' \geq \mathsf{max_lazy}$. Therefore, LZ77 outputs a back-reference `<L`k`,D*>` and moves the pointer to $S[k+1]$ in the query. Continuing this process, we conclude that LZ77 outputs back-references for $S\left[1 : k\right], S\left[k+1 : 2k\right], \ldots, S\left[(m-1)k+1 : mk\right]$, likely all in dict_2, and moves the pointer to $S[mk+1]$, just past S.
- Otherwise, if $ck[1] \neq g$, then LZ77 finds a match for P and moves the pointer to g in the query Q. Next, by condition 1, LZ77 finds a match for $S\left[0 : k'-1\right] = g\|S\left[1 : k'-1\right]$ in the sliding window, likely in dict_1. By condition 1, the match cannot be extended and lazy matching is either disabled or not performed because the match is of length $k' \geq \mathsf{max_lazy}$.[73] Therefore, LZ77 outputs a back-reference `<L`k'`,D*>` and moves the pointer to $S[k']$ in the query. Next, by condition 3, LZ77 cannot find a match starting at $S[k'], \ldots, S[k-1]$; therefore, LZ77 outputs $S\left[k' : k-1\right]$ verbatim and moves the pointer to $S[k]$. Continuing this process, we know that LZ77 outputs back-references for $g\|S\left[1 : k'-1\right], S\left[k : k+k'-1\right], \ldots, S\left[(m-2)k : (m-2)k+k'-1\right]$, likely all in dict_1, does not find matches for $S\left[k'+1 : k-1\right], \ldots, S\left[(m-2)k+k'+1 : (m-1)k-1\right]$ and moves the pointer to $S[(m-1)k]$. Finally, by applying condition 2, we conclude that LZ77 finds $S\left[(m-1)k : mk-1\right]$, likely in dict_1, which cannot be extended or improved with lazy matching, outputs `<L`k`,D*>`, and moves the pointer to $S[mk]$.

Fast match chain: performance. If $ck[1] = g$, then the LZ77 output consists of $m+1$ back-references, most of which have length k. Otherwise, the LZ77 output consists of $m+1$ back-references, most of which have length k', as well as $(m-1)(k-k')+1$ literals. That is, the LZ77 output has $(m-1)(k-k')+1$ more symbols when $ck[1] \neq g$. This should lead to a similar difference in the compressed lengths. If we make a simplifying assumption that Huffman coding encodes each back-reference to about the same length in both cases, then we know the compressed length difference should be roughly proportional to $m(k-k')$,

[73] If we allow $k' < \mathsf{max_lazy}$, then LZ77 likely finds a better match $S\left[1 : k\right]$ in dict_2 and moves the pointer to $S[k+1]$ in the query, and from then on LZ77 behaves as in the case where $ck[1] = g$.

where $k - k' \leq \mathsf{min_match} - 2$; the exact difference depends on Huffman coding, and in particular the codeword lengths for the literals, but it should be close to $m(k-k')$ bytes. The query length ℓ_Q is about $m(2k+k'+2)$ bytes with one-byte dictionary separators and an empty filler. In comparison, under the same parameters m and k, a distance chain is slightly longer (around $m(3k+2)$ bytes) but produces a smaller estimated compressed length difference (around $m/4$ bytes).

Fast match chain: reverse amplification. Reversing the direction of amplification is not as straightforward as in distance-based chaining; here we only give the byte string sequences for dict_1 and dict_2 in our construction for reverse amplification. In this construction, we require that $k' = k - \mathsf{min_match} + 2$. The dictionary string dict_1 represents the sequence of m byte strings

$$(S[1:k+k'-2], S[2k-1:2k+k'-2], \ldots, S[(m-1)k-1:(m-1)k+k'-2], S[(m-1)k+k'-1:mk])$$

of total length $(m-2)k'+2k$, and dict_2 represents the sequence of m byte strings $\{S[(i-1)k:ik-1]\}_{i\in[m]}$ of total length mk. In our example setting, for $g = \mathtt{1}$, $m = 5$, $k = 4$ and $k' = 3$, the query can be

```
-ABCDE-GHI-JLM-OPQ-RST--1ABC-DEFG-HIJK-LMNO-PQRS-secret=1ABCDEFGHIJKLMNOPQRST
```

Fast match chain: doubling. We briefly describe our construction for reducing the length of a fast match chain query using the doubling strategy, which is almost identical to the doubling strategy for distance-based chaining. For simplicity, we only consider the normal direction of amplification here; a similar doubling construction for reverse amplification is available in our implementation.

Assume $m = 2m' - 1$ for some $m' \in \mathbb{N}_+$. Let S' be a byte string of length $m'k$. We construct the distance chain S of length $\ell_Q = mk$ as

$$S'[1:k] \| S'[k+1:2k] \| S'[k+1:2k] \| \cdots \| S'[(m'-1)k+1:m'k] \| S'[(m'-1)k+1:m'k].$$

The dictionary string dict_2 represents the sequence of byte strings $(S[(i-1)k+1:ik])_{i\in[m]}$, which contains at least $(m-1)/2$ repeated entries in the sequence; one can construct dict_2 using around $(m+1)(k+1)/2$ bytes, roughly halving the space normally required. Doubling also enlarges the differences in the LZ77 back-reference distances. As with distance-based chaining, we can decrease the failure probability by selecting S' such that for each $i_1 \in [m'-1]$, $S'[i_1 k] \neq S'[(i_1+1)k]$, and for each $i_2 \in [m'-2]$, $S'[i_2 k+1] \neq S'[(i_2+1)k+1]$.

In our example setting, for $g = \mathtt{1}$, $m = 5$, $k = 4$ and $k' = 3$, a query with doubling can be

```
-1AB-DEF-HEF-HIJ-LIJK--ABCD-EFGH-IJKL-secret=1ABCDEFGHEFGHIJKLIJKL
```

for $S' = \mathtt{ABCDEFGHIJKL}$.

Finally, we note that it is also possible to reduce the query size by overlapping, another optimisation strategy we discussed for distance-based chaining, which we also implemented for the fast match chain, but it is generally less effective and more error-prone than doubling.

Lazy match chain: construction. Because most DEFLATE implementations perform lazy matching with reasonably large $\mathsf{max_lazy}$ by default, it is more interesting to study match-based chaining with lazy matching enabled, for which case we design a lazy match chain as follows.

Assume lazy matching is enabled and $\mathsf{max_lazy} \geq 4$. We require that $k \geq \mathsf{min_match} + 1$. If $k \geq \mathsf{min_match} + 2$, then we require that $k \leq \mathsf{max_lazy} + 1$. Let $k' \in [k-1]$, where $k' \geq \mathsf{min_match}$. Let $\delta = k - k' + 2$. If $\delta \geq \mathsf{min_match}$, then we require that $k = k' + 1$ and $k' > 3$.

A lazy match chain S is a byte string of length $\ell_S = mk$. The dictionary string dict_1 represents the sequence of m byte strings

$$(S[0:k-2], S[k:k+k'-1], \ldots, S[(m-2)k:(m-2)k+k'-1], S[(m-1)k:mk-1])$$

of total length $(m-2)k' + 2k - 1$, and dict_2 represents the sequence of m byte strings

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

of total length $mk - 1$.

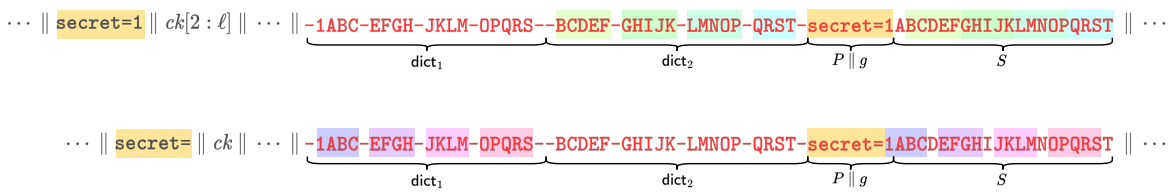


Fig. 2: Illustration of match-based chaining. The query Q is in red and the likely LZ77 matches are highlighted. It shows the effect of amplification constructed as a lazy match chain if the guess for $ck[1]$ is correct (top) and when it is not (bottom).

In our example setting, for $g = \mathtt{1}$, $m = 4$, $k = 5$ and $k' = 4$, the query can be

$$\mathtt{-1ABC-EFGH-JKLM-OPQRS--BCDEF-GHIJK-LMNOP-QRST-secret=1ABCDEFGHIJKLMNOPQRST}$$

where we use - as separators in dict_1 and dict_2, and we set the filler string filler to the empty string ε.

We illustrate the behaviour of LZ77 in our example in Fig. 2.

If $ck[1] = g$, then LZ77 encodes $\mathsf{gadgets} = P\|g\|S$ in the query as

$$\mathtt{<L}(\ell_P + 1)\mathtt{,D*>}\|S[1]\|\mathtt{<L}k\mathtt{,D*>}\|\cdots\|\mathtt{<L}k\mathtt{,D*>}\|\mathtt{<L}(k-1)\mathtt{,D*>},$$

where the back-references and literals correspond to

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

which are $\mathtt{secret=1}$, $\mathtt{A}$, $\mathtt{BCDEF}$, $\mathtt{GHIJK}$, $\mathtt{LMNOP}$, $\mathtt{QRST}$ in our example. All back-references except the first one are expected to point to a byte string in dict_2.

Otherwise, if $ck[1] \neq g$, then LZ77 encodes $\mathsf{gadgets} = P\|g\|S$ in the query as

$$\begin{aligned}&\mathtt{<L}\ell_P\mathtt{,D*>}\|\mathtt{<L}(k-1)\mathtt{,D*>}\|S[k-1]\|\mathtt{<L}k'\mathtt{,D*>}\|S\left[k+k' : 2k-1\right]\|\cdots\|\\&\mathtt{<L}k'\mathtt{,D*>}\|S\left[(m-2)k+k' : (m-1)k-1\right]\|\mathtt{<L}k\mathtt{,D*>}\|S[mk],\end{aligned}$$

where the back-references and literals correspond to

$$\begin{aligned}&P, S\left[0 : k-2\right], S[k-1], S\left[k : k+k'-1\right], S\left[k+k' : 2k-1\right], \ldots,\\&S\left[(m-2)k : (m-2)k+k'-1\right], S\left[(m-2)k+k' : (m-1)k-1\right], S\left[(m-1)k : mk-1\right], S[mk],\end{aligned}$$

which are $\mathtt{secret=}$, $\mathtt{1ABC}$, $\mathtt{D}$, $\mathtt{EFGH}$, $\mathtt{I}$, $\mathtt{JKLM}$, $\mathtt{N}$, $\mathtt{OPQRS}$, $\mathtt{T}$ in our example. All back-references except the first one are expected to point to a byte string in dict_1.

Lazy match chain: requirements. For the lazy match chain to work correctly, we require that:

1. The byte string $S\left[0 : k-2\right]$ in dict_1 can be found by LZ77 in the sliding window, but $S\left[1 : k-1\right]$ and $S\left[k-1 : k+2\right]$ do not appear in the sliding window.[74] If $k \geq \mathsf{min_match} + 2$, then $S\left[1 : k-2\right]$ can also be found by LZ77 in the sliding window.
2. For each $i \in [m-2]$, $S\left[ik : ik+k'-1\right]$ in dict_1 can be found by LZ77 in the sliding window, but $S\left[ik : ik+k'\right]$ and $S\left[ik+1 : ik+k'+1\right]$ do not appear in the sliding window.
3. If $\delta < \mathsf{min_match}$, then for each $i \in [m-2]$ and $j \in \mathbb{N}_+$ such that $ik+k' \leq j \leq (i+1)k-1$, $S\left[j : j+\mathsf{min_match}-1\right]$ does not appear in the sliding window.
4. If $\delta \geq \mathsf{min_match}$, then for each $i \in [m-2]$, the byte string $S\left[ik+k' : (i+1)k+1\right]$, a substring of $S\left[ik+2 : (i+1)k+1\right]$ in dict_2, can be found in the sliding window, but $S\left[ik+k' : (i+1)k+2\right]$ does not appear in the sliding window.
5. The byte string $S\left[(m-1)k : mk-1\right]$ in dict_1 can be found by LZ77 in the sliding window, but $S\left[(m-1)k : mk\right]$ and $S\left[(m-1)k+1 : mk+1\right]$ do not appear in the sliding window.
6. For each $i \in [m-1]$, $S\left[(i-1)k+2 : ik+1\right]$ in dict_2 can be found by LZ77 in the sliding window, but $S\left[(i-1)k+2 : ik+2\right]$ and $S\left[(i-1)k+3 : ik+3\right]$ do not appear in the sliding window.
7. The byte string $S\left[(m-1)k+2 : mk\right]$ in dict_2 can be found by LZ77 in the sliding window, but $S\left[(m-1)k+2 : mk+1\right]$ and $S\left[(m-1)k+3 : mk+2\right]$ do not appear in the sliding window.[75]

[74] Note that $S\left[k-1 : k+1\right]$ likely appears in the sliding window as a substring of $S\left[2 : k+1\right]$ in dict_2.

[75] Note that $(m-1)k+3 < mk-1$ by our assumptions.

Our analysis utilises conditions 1, 6, 7 for the case where $ck[1] = g$ and conditions 1, 2, 3, 4, 5 for the case where $ck[1] \neq g$. Intuitively, conditions 1, 5, 7 control the behaviour of LZ77 at the first and the last segments of the chain, while conditions 2, 3, 4, 6 apply more broadly to most segments of the chain. Note that, depending on the parameter δ, either condition 3 or 4 is trivially satisfied; the distinction is because LZ77 behaves differently depending on whether $\delta \geq \mathsf{min_match}$ or not, which is carefully handled in our analysis.

Readers can check that no requirements conflict with each other within the parameter range we specified. Under reasonable parameters, the above conditions likely hold if one constructs the lazy match chain S by sampling each byte independently and uniformly at random from a large subset of Σ, but other constructions are also feasible.

Lazy match chain: analysis. We now justify the output of LZ77 for $\mathsf{gadgets} = P\|g\|S$. We assume the LZ77 pointer is initially at the first byte of P.

- If $ck[1] = g$, then LZ77 finds a match for $P\|g$ in the sliding window and moves the pointer to $S[1]$ in the query Q. Next, LZ77 outputs the literal $S[1]$ and a back-reference for $S[2 : k+1]$, likely in dict_2, and moves the pointer to $S[k+2]$, because:
  - If $k < \mathsf{min_match}+2$, then any byte string starting at $S[1]$ of length at least $\mathsf{min_match}$ contains $S[1 : k-1]$ as a substring and therefore does not appear in the sliding window by condition 1, so LZ77 does not find a match starting at $S[1]$, outputs $S[1]$, and moves the pointer to $S[2]$. There, by condition 6, LZ77 finds $S[2 : k+1]$ in the sliding window, likely in dict_2, which cannot be extended or improved by lazy matching.
  - If $k \geq \mathsf{min_match}+2$, then by condition 1, LZ77 finds a match for $S[1 : k-2]$ of length $k-2$ in the sliding window, which cannot be extended. Because now $k \leq \mathsf{max_lazy}+1$ by assumption, LZ77 tries to improve the match by lazy matching, and, by condition 6, finds $S[2 : k+1]$ in the sliding window, likely in dict_2, which cannot be extended or improved by lazy matching.

  At position $S[k+2]$, by condition 6, LZ77 finds $S[k+2 : 2k+1]$ in the sliding window, likely in dict_2, which cannot be extended or improved by lazy matching. Therefore, LZ77 outputs a back-reference for $S[k+2 : 2k+1]$ and moves the pointer to $S[2k+2]$. Continuing this process, we know that LZ77 outputs back-references for $S[k+2 : 2k+1], \ldots, S[(m-2)k+2 : (m-1)k+1]$, likely all in dict_2, and moves the pointer to $S[(m-1)k+2]$. By condition 7, LZ77 finds a match for $S[(m-1)k+2 : mk]$ of length $k-1 \geq \mathsf{min_match}$, likely in dict_2, which cannot be extended or improved by lazy matching, and moves the pointer to $S[mk+1]$, just past S.
- Otherwise, if $ck[1] \neq g$, then LZ77 finds a match for P and moves the pointer to g in the query Q. Next, by condition 1, LZ77 finds a match for $S[0 : k-2] = g\|S[1 : k-2]$ of length $k-1 \geq \mathsf{min_match}$ in the sliding window, likely in dict_1. By condition 1, the match cannot be extended or improved by lazy matching. Therefore, LZ77 outputs a back-reference for $S[0 : k-2]$ and moves the pointer to $S[k-1]$ in the query. Next, LZ77 outputs a literal $S[k-1]$ and a back-reference for $S[k : k+k'-1]$, likely in dict_1, and moves the pointer to $S[k+k']$, because:
  - If LZ77 does not find a match starting at $S[k-1]$, then LZ77 outputs $S[k-1]$ and moves the pointer to $S[k]$, where by condition 2, LZ77 finds a longer match $S[k : k+k'-1]$ of length k', likely in dict_1, which cannot be extended or improved by lazy matching.
  - If LZ77 finds a match starting at $S[k-1]$, then, by condition 1, the match does not contain $S[k-1 : k+2]$ as a substring and is therefore at most 3 bytes, so we know $\mathsf{min_match} = 3$, and thus $\delta = k - k' + 2 \geq 3 = \mathsf{min_match}$, which by assumption implies $k' > 3$. Because $\mathsf{max_lazy} > 3$ by assumption, LZ77 performs lazy matching and, by condition 2, finds a longer match $S[k : k+k'-1]$ of length k', likely in dict_1, which cannot be extended or improved by lazy matching.

  At position $S[k+k']$, LZ77 outputs $k - k' = \delta - 2$ literals $S[k+k' : 2k-1]$ and a back-reference for $S[2k : 2k+k'-1]$, likely in dict_1, and moves the pointer to $S[2k+k']$, because:
  - If $\delta < \mathsf{min_match}$, then by condition 3, LZ77 cannot find any match at $S[k+k'], \ldots, S[2k-1]$, outputs the literals $S[k+k' : 2k-1]$, and moves the pointer to $S[2k]$. There, by condition 2, LZ77 finds a match $S[2k : 2k+k'-1]$, likely in dict_1, which cannot be extended or improved by lazy matching.
  - If $\delta \geq \mathsf{min_match}$, then by assumption, $k = k'+1$, and $\delta = k - k' + 2 = 3$. By condition 4, LZ77 finds a match for $S[k+k' : 2k+1]$ of length $k - k' + 2 = 3 \geq \mathsf{min_match}$ in the sliding window, likely in dict_2, which cannot be extended. Because $\mathsf{max_lazy} > 3$, LZ77 performs lazy matching by looking for any match starting at the next byte, which is $S[2k]$. By condition 2, LZ77 finds a longer match $S[2k : 2k+k'-1]$ of length $k' > 3$, likely in dict_1, which cannot be extended or improved by lazy matching.

Continuing this process, we know that LZ77 outputs $\delta - 2$ literals $S[ik + k' : (i+1)k - 1]$ and a back-reference for $S[(i+1)k : (i+1)k + k' - 1]$ for each $i \in [m-3]$, and moves the pointer to $S[(m-2)k + k']$. The $m-3$ back-references are likely all in dict_1. At $S[(m-2)k + k']$, one can use a very similar argument as above to deduce that LZ77 outputs literals $S[(m-2)k + k' : (m-1)k - 1]$ and a back-reference for $S[(m-1)k : mk - 1]$, likely in dict_1, and moves the pointer to $S[mk]$:

- If $\delta < \mathsf{min_match}$, then by condition 3, LZ77 cannot find any match at $S[(m-2)k + k'], \ldots, S[(m-1)k - 1]$, outputs the literals $S[(m-2)k + k' : (m-1)k - 1]$, and moves the pointer to $S[(m-1)k]$. There, by condition 5, LZ77 finds a match $S[(m-1)k : mk - 1]$, likely in dict_1, which cannot be extended or improved by lazy matching.
- If $\delta \geq \mathsf{min_match}$, then by assumption, $k = k' + 1$, and $\delta = k - k' + 2 = 3$. By condition 4, LZ77 finds a match for $S[(m-2)k + k' : (m-1)k + 1]$ of length $k - k' + 2 = 3 \geq \mathsf{min_match}$ in the sliding window, likely in dict_2, which cannot be extended. Because $\mathsf{max_lazy} > 3$, LZ77 performs lazy matching by looking for any match starting at the next byte, which is $S[(m-1)k]$. By condition 5, LZ77 finds a longer match $S[(m-1)k : mk - 1]$ of length $k > 3$, likely in dict_1, which cannot be extended or improved by lazy matching.

Lazy match chain: performance. If $ck[1] = g$, then the LZ77 output consists of $m+1$ back-references, most of which have length k, as well as one literal. Otherwise, the LZ77 output consists of $m+1$ back-references, most of which have length k', as well as $(m-1)(k-k') + 2$ literals. That is, the LZ77 output has $(m-1)(k-k') + 1$ more symbols when $ck[1] \neq g$. This should lead to a similar difference in the compressed lengths. Similar to the fast match chain, if we assume that Huffman coding encodes each back-reference to about the same length in both cases, then the compressed length difference should be roughly proportional to $m(k-k')$, where $k - k' \leq \max\{1, \mathsf{min_match} - 3\}$ by our assumptions. The length of the query is about $m(2k + k' + 2)$ bytes with one-byte dictionary separators and an empty filler.

The performance of a lazy match chain is very similar to a fast match chain with the same parameters. However, a lazy match chain has more complex requirements and is therefore more likely to fail in general. Note also that, based on our assumptions, if $\mathsf{min_match} = 3$, the minimum value of the segment length k is 5 for lazy match chains and 4 for fast match chains.

Lazy match chain: reverse amplification. We briefly describe our construction for reverse amplification by giving the byte string sequences for dict_1 and dict_2. The dictionary string dict_1 represents the sequence of m byte strings

$$(S[2 : k + k' - 3], S[2k - 2 : 2k + k' - 3], \ldots, S[(m-1)k - 2 : (m-1)k + k' - 3], S[(m-1)k + k' - 2 : mk])$$

of total length $(m-2)k' + 2k - 1$, and dict_2 represents the sequence of m byte strings

$$(S[0 : k - 2], S[k : 2k - 1], \ldots, S[(m-2)k : (m-1)k - 1], S[(m-1)k : mk - 1])$$

of total length $mk - 1$. In our example setting, for $g = \mathtt{1}$, $m = 4$, $k = 5$ and $k' = 4$, the query can be

```
-BCDEF-HIJK-MNOP-QRST--1ABC-EFGHI-JKLMN-OPQRS-secret=1ABCDEFGHIJKLMNOPQRST
```

Lazy match chain: doubling. Finally, we briefly describe our construction for reducing the length of a lazy match chain query using the doubling strategy. For simplicity, we only consider the normal direction of amplification here; a similar doubling construction for reverse amplification is available in our implementation.

Assume $m = 2m'$ for some $m' \in \mathbb{N}_+$. Let S' be a byte string of length $m'k + 1$. We construct the chain S of length $\ell_Q = mk$ as

$$S'[1] \| S'[2 : k+1] \| S'[2 : 2k+1] \| \cdots \| S'[(m'-2)k + 2 : m'k + 1] \| S'[(m'-1)k + 2 : m'k].$$

The dictionary string dict_2 represents the sequence with at least $m/2 - 1$ repeated entries; in addition, the last entry $S'[(m'-1)k + 2 : m'k]$ in dict_2 is a substring of $S'[(m'-1)k + 2 : m'k + 1]$, another entry in dict_2, and can be omitted. Therefore, one can construct dict_2 using around $m(k+1)/2$ bytes, roughly halving the space normally required. Doubling also enlarges the differences in the LZ77 back-reference distances. Similar to other doubling constructions, we can decrease the failure probability by selecting S' such that for each $i \in [m']$, $S'[(i-1)k + 1] \neq S'[ik + 1]$.

In our example setting, for $g = \mathtt{1}$, $m = 4$, $k = 5$ and $k' = 4$, a query with doubling can be

```
-1ABC-EFBC-EFGH-JKGHI--BCDEF-GHIJK-secret=1ABCDEFBCDEFGHIJKGHIJ
```

7 Collision-based amplification

In this section, we introduce a technique that we refer to as *collision-based amplification*. Our technique amplifies compressed length differences in DEFLATE implementations by exploiting the hash table in LZ77 for finding matches in the sliding window. More specifically, a collision-based amplification query contains a large number of substrings with the same hash value, such that when LZ77 scans through the query, the corresponding hash chain becomes too long and older entries on that chain are no longer considered by LZ77. The query is carefully designed such that LZ77 cannot find back-references at certain positions in the query despite suitable matches being in the sliding window, and as a result LZ77 finds very different back-references depending on whether the attacker's guess is correct.

Unlike the techniques we described in Section 6, the compressed length difference from collision-based amplification is not limited by the sliding window size $\mathsf{window_size}$ or the prefix length ℓ_P as long as they are not too small, and it grows about linearly to the query length.[76] This may come as a surprise given that the additive sensitivity of LZ77 in [ZL77] has been shown to be $\mathcal{O}\left(\mathsf{window_size}^{2/3} \log n\right)$ [BLG25], where n is the length of the uncompressed plaintext, whereas our result shows that the additive sensitivity of LZ77 in most DEFLATE implementations is $\Omega(n)$.[77] This discrepancy stems from the use of the hash table for LZ77 in DEFLATE, and it highlights the need to study real-world compressors for compression side-channel attacks and defences.

Note that the collision-based amplification technique we describe in this section does not target the general DEFLATE algorithm, because it depends on some details of the LZ77 hash table that are unspecified in DEFLATE, in particular the (non-cryptographic) hash function used. That said, our technique applies to zlib as well as all implementations we examined in Section 3.5, and we expect most DEFLATE compressors to be susceptible to variants of collision-based amplification. More generally, we can potentially amplify the compressed length differences in other LZ77-based compressors, such as LZ4 and brotli, by exploiting their implementations of hash tables or other data structures they use for finding matches in the sliding window.

We will describe our constructions for $\mathsf{deflate_slow}$ and $\mathsf{deflate_fast}$, two compression strategies in zlib that are used in practically all DEFLATE implementations. As in Section 6, we present the collision-based amplification technique based on the scenario where an attacker wants to check whether the first byte of the secret $ck[1]$ is equal to their guess $g \in \Omega$. Our constructions in this section all follow the general framework in Section 6.3. In particular, the query Q is of the form $\mathsf{dict}_1 \| \mathsf{filler} \| \mathsf{dict}_2 \| P \| g \| S$. As with our description of the chaining techniques in Section 6.3, we abuse notation to let $S[0]$ denote the byte immediately before S in the plaintext (i.e. g), and for each $i \in \mathbb{N}_+$, we let $S[\ell_S + i]$ denote the i-th byte located after S in the plaintext, or a special end symbol if there are fewer than i bytes after S.

7.1 Collisions

The central part of collision-based amplification is to create collisions in the LZ77 hash table, which requires examining the hash function used in the DEFLATE implementation we target. As we have seen in Sections 3.4 and 3.5, DEFLATE implementations typically use simple hash functions with small domains and ranges, which allow the attacker to efficiently enumerate all possible collisions via exhaustive search. Furthermore, for LZ77 hash functions, it is generally feasible to find collisions that satisfy some special properties, e.g. double-colliding (Definition 5), which are useful for our attack.

We first give a formal definition of DEFLATE hash functions and collisions.

[76] For unlimited amplification, at least one of the following assumptions about the *flush mode* of the compressor must hold: (i) the flush mode is `Z_NO_FLUSH` or `Z_FINISH` in zlib or their counterparts in other DEFLATE implementations, or (ii) the flush mode is not `Z_FULL_FLUSH` in zlib or its counterparts in other DEFLATE implementations, and at least `MIN_LOOKAHEAD` (default 262) bytes of input are always available when the LZ77 pointer is within the query, except at the end of the query. Normally (i) is true when using zlib and other DEFLATE implementations, but some applications may choose other flush modes for better performance, in which case one needs to check if (ii) holds (for example by checking the sizes of the input and output buffers). If neither assumption holds, then the compressor may break very long queries in the middle, possibly making them ineffective. We do not discuss flush modes in this paper for simplicity, and we refer interested readers to https://www.bolet.org/~pornin/deflate-flush.html and https://zlib.net/manual.html for more details.

[77] Some DEFLATE implementations may have (very large) upper bounds on the input length. For example, while zlib can compress any amount of data correctly, the `compress()` utility function in zlib may limit the input to at most 4 GiB; see https://zlib.net/zlib_faq.html#faq32.

Definition 1 (DEFLATE hash function). *Let* $\mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$*, where* $\mathsf{min_match} \geq 2$*. A DEFLATE hash function with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$*, or a hash function in short, is a function*

$$\mathsf{H} : \{0, \ldots, 255\}^{\mathsf{min_match}} \to \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\}.$$

Definition 2 (Rolling hash function). *Let* $\mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$*, where* $\mathsf{min_match} \geq 2$*. A DEFLATE hash function* H *with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$ *is rolling, if there exists a function*

$$f_{\mathsf{H}} : \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\} \times \{0, \ldots, 255\} \to \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\},$$

such that for every byte string T *of length* $\mathsf{min_match} + 1$*, it holds that*

$$f_{\mathsf{H}}\left(\mathsf{H}(T\left[1 : \mathsf{min_match}\right]), T[\mathsf{min_match} + 1]\right) = \mathsf{H}(T\left[2 : \mathsf{min_match} + 1\right]).$$

Informally, a rolling hash function allows DEFLATE to compute the hash value of a $\mathsf{min_match}$-byte substring of the input from the hash value of the substring starting at its previous byte and the last byte of the current substring. An example rolling hash function is the hash function in zlib (Section 3.4).[78]

Definition 3 (Colliding set). *Let* $k, \mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$*, where* $\mathsf{min_match} \geq 2$*, let* $\Sigma \subseteq \{0, \ldots, 255\}$*,* $v \in \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\}$*, and let* H *be a DEFLATE hash function with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$*. A set of* k *byte strings* $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^{\geq \mathsf{min_match}}$ *is colliding at* v *with regard to* H *on* Σ*, if for every* $i \in [k]$*,* $\mathsf{H}(T_i\left[1 : \mathsf{min_match}\right]) = v$*.*
In addition, we say $\{T_1, T_2, \ldots, T_k\}$ *is compact-colliding if* $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^{\mathsf{min_match}}$*.*

By definition, starting from a colliding set on Σ, one can construct another colliding set on Σ of any finite size, either by taking a subset of the original colliding set or by concatenating the strings in the original colliding set with different byte strings in Σ^*. Finding the largest compact-colliding sets on Σ takes at most $|\Sigma|^{\mathsf{min_match}}$ hash evaluations, typically no more than 2^{32}, and we may further reduce the number of hash evaluations by exploiting the mathematical structure of the DEFLATE hash function in question.

By the pigeonhole principle, for every DEFLATE hash function with the parameters $\mathsf{min_match}$ and $\mathsf{hash_bits}$ and for every $\Sigma \subseteq \{0, \ldots, 255\}$, there exists a compact-colliding set on Σ of at least $\left\lceil |\Sigma|^{\mathsf{min_match}} / 2^{\mathsf{hash_bits}} \right\rceil$ strings; the maximum size of a compact-colliding set may be higher than this bound, especially if Σ is structured. For example, the maximum size of a compact-colliding set of alphanumeric strings is 27 in zlib with $\mathsf{min_match} = 3$ and $\mathsf{hash_bits} = 15$ (e.g. at hash value 20081, whereas the bound is $\lceil 62^3/2^{15} \rceil = 8$), 489 in zlib-cloudflare with $\mathsf{min_match} = 4$ and $\mathsf{hash_bits} = 15$ (at hash value 12663, whereas the bound is $\lceil 62^4/2^{15} \rceil = 451$), and 266 in zlib-ng with $\mathsf{min_match} = 4$ and $\mathsf{hash_bits} = 16$ (at hash value 38110, whereas the bound is $\lceil 62^4/2^{16} \rceil = 226$).

We also define a function that counts the number of collisions at a specific hash value within a given byte string, which we use to simplify the description and analysis of our technique in Sections 7.3 and 7.4.

Definition 4 (Collision counter). *Let* $k, \mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$*, where* $\mathsf{min_match} \geq 2$*, and let* H *be a DEFLATE hash function with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$*. A collision counter for* H *is a function*

$$cc_{\mathsf{H}} : \{0, \ldots, 255\}^* \times \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\} \to \mathbb{N},$$

where for each byte string $W \in \{0, \ldots, 255\}^*$ *and hash value* $v \in \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\}$*,*

$$cc_{\mathsf{H}}(W, v) = |\{i \mid i \in [|W| - \mathsf{min_match} + 1] \wedge \mathsf{H}\left(W\left[i : i + \mathsf{min_match} - 1\right]\right) = v\}|.$$

Intuitively, for DEFLATE implementations that perform lazy matching, the attacker needs to control the behaviour of LZ77 for both normal and lazy matching by creating collisions in the LZ77 hash table, which means the attacker would need to find a colliding set of strings that is still colliding after removing the first byte of each string. To this end, we consider a stronger collision property called *double-colliding*, which we define below.

[78] Because we defined the zlib hash function in Section 3.4 as (implicitly) a family of DEFLATE hash functions indexed by $\mathsf{hash_bits}$, a more precise statement is that the zlib hash function is rolling for every fixed $\mathsf{hash_bits}$.

Definition 5 (Double-colliding set). *Let* $k, \mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$*, where* $\mathsf{min_match} \geq 2$*, let* $\Sigma \subseteq \{0, \ldots, 255\}$*,* $v \in \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\}$*, and let* H *be a DEFLATE hash function with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$*. A set of* k *byte strings* $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^{>\mathsf{min_match}}$ *is double-colliding at* v *with regard to* H *on* Σ*, if for every* $i \in [k]$*, it holds that*

$$\mathsf{H}(T_i\,[1 : \mathsf{min_match}]) = \mathsf{H}(T_i\,[2 : \mathsf{min_match} + 1]) = v.$$

In addition, we say $\{T_1, T_2, \ldots, T_k\}$ *is compact-double-colliding if* $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^{\mathsf{min_match}+1}$.

Note that a double-colliding set at some hash value v with regard to H is naturally also colliding at v with regard to H.

Similarly, one can construct a double-colliding set on Σ of any finite size from a compact-double-colliding set on Σ. Finding the largest compact-double-colliding sets on Σ takes at most $|\Sigma|^{\mathsf{min_match}+1}$ hash evaluations. This is typically no more than 2^{40}, or much fewer if the DEFLATE hash function has some special properties, e.g. it is rolling.

It is clear that for every byte c, a singleton set consisting of a string $cc\ldots c$ of length $\mathsf{min_match} + 1$ is compact-double-colliding; however, one should not expect to find a compact-double-colliding set of at least two strings for every hash function, especially if the hash function is not rolling or $\mathsf{hash_bits}$ is large. For a hash function H modelled as a random oracle, at each possible hash value v, the expected size of the largest compact-double-colliding set at v on Σ is

$$\sum_{\substack{T \in \Sigma^{\mathsf{min_match}+1} \\ T[1:\mathsf{min_match}] \neq T[2:\mathsf{min_match}+1]}} \frac{1}{2^{2 \cdot \mathsf{hash_bits}}} + \sum_{\substack{T \in \Sigma^{\mathsf{min_match}+1} \\ T[1:\mathsf{min_match}] = T[2:\mathsf{min_match}+1]}} \frac{1}{2^{\mathsf{hash_bits}}}$$
$$= \frac{|\Sigma|}{2^{2 \cdot \mathsf{hash_bits}}} \left(|\Sigma|^{\mathsf{min_match}} + 2^{\mathsf{hash_bits}} - 1 \right).$$

For rolling hash functions, we are usually able to find relatively large compact-double-colliding sets. In particular, for any compact-colliding set that contains a string $cc\ldots c$ of length $\mathsf{min_match}$ for some byte c, we can construct a compact-double-colliding set of the same size by appending the byte c to every string in the compact-colliding set.[79] For example, the maximum size of a compact-double-colliding set of alphanumeric strings is 27 for zlib with $\mathsf{min_match} = 3$ and $\mathsf{hash_bits} = 15$ (e.g. at hash value 20081, which contains the string `QQQQ`), same as the maximum size of a compact-colliding set of alphanumeric strings. We show this formally in Theorem 6.

Theorem 6. *Let* $k, \mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$*, where* $\mathsf{min_match} \geq 2$*, let* $\Sigma \subseteq \{0, \ldots, 255\}$*,* $c \in \Sigma$*,* $v \in \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\}$*, and let* H *be a DEFLATE hash function with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$*, where* H *is rolling. If a set of* k *byte strings* $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^{\mathsf{min_match}}$ *is compact-colliding at* v *with regard to* H *on* Σ*, and* $\mathsf{H}(c^{\mathsf{min_match}}) = v$*, then the set* $\{T_1 \| c, T_2 \| c, \ldots, T_k \| c\} \subseteq \Sigma^{\mathsf{min_match}+1}$ *of size* k *is compact-double-colliding at* v *with regard to* H *on* Σ*.*

Proof. By Definition 2, there exists $f_{\mathsf{H}} : \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\} \times \{0, \ldots, 255\} \to \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\}$, such that for every $T \subseteq \Sigma^{\mathsf{min_match}+1}$,

$$f_{\mathsf{H}}\left(\mathsf{H}(T\,[1 : \mathsf{min_match}]), T[\mathsf{min_match} + 1]\right) = \mathsf{H}(T\,[2 : \mathsf{min_match} + 1]).$$

Applying the above equation to the byte string $c^{\mathsf{min_match}+1}$ yields $f_{\mathsf{H}}(\mathsf{H}(c^{\mathsf{min_match}}), c) = \mathsf{H}(c^{\mathsf{min_match}})$, and since $\mathsf{H}(c^{\mathsf{min_match}}) = v$, we obtain that $f_{\mathsf{H}}(v, c) = v$. Therefore, for every $i \in [k]$,

$$\mathsf{H}\left(T_i\,[2 : |T_i|] \,\|c\right) = \mathsf{H}\left(T_i\,[2 : \mathsf{min_match}] \,\|c\right) = f_{\mathsf{H}}\left(\mathsf{H}(T_i), c\right) = f_{\mathsf{H}}(v, c) = v.$$

We conclude the proof by noting that $\mathsf{H}(T_i) = v$ for every $i \in [k]$ by definition. □

For hash functions that are not rolling, compact-double-colliding sets are generally much smaller. The maximum size of a compact-double-colliding set of alphanumeric strings is 5 for zlib-ng (e.g. at hash value 44587, with the set {`jij7l`, `jk6Pr`, `jEFAx`, `jMi22`, `jO5K8`}), 6 for zlib-cloudflare (e.g. at hash value 7197, with the set {`id2bs`, `lqq5v`, `Ibi5i`, `QmzxL`, `W0VRJ`, `1FFZb`}) and 4 for zlib-go (e.g. at hash value 46484, with the set {`aNBdb`, `nY1Qb`, `TCSwb`, `1Vs9b`}). A notable exception is chromium-zlib, for which there exists a compact-double-colliding set of 171 alphanumeric strings (at hash value 2185, including `eeeee`).[80]

[79] The constructed set is also strongly compact-double-colliding, which we define later in Definition 7.

[80] We assume in this paragraph that all compressors use their default parameters for $\mathsf{min_match}$ and $\mathsf{hash_bits}$.

In addition to double-colliding, we discuss two alternative collision properties that can be used for collision-based amplification when lazy matching is enabled.

An earlier version of this work used a stronger variant of double-colliding for collision-based amplification [Son24], which we refer to here as *strongly double-colliding*.

Definition 7 (Strongly double-colliding set). *Let* k, $\mathsf{min_match}$, $\mathsf{hash_bits} \in \mathbb{N}_+$, *where* $\mathsf{min_match} \geq 2$, *let* $\Sigma \subseteq \{0, \ldots, 255\}$, $v \in \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\}$, *and let* H *be a DEFLATE hash function with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$. *A set of* k *byte strings* $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^{>\mathsf{min_match}}$ *is strongly double-colliding at* v *with regard to* H *on* Σ, *if* $\{T_1, T_2, \ldots, T_k\}$ *is double-colliding at* v *with regard to* H, *and there exists some byte* $c \in \Sigma$, *such that* $\mathsf{H}(c^{\mathsf{min_match}}) = v$.[81]
In addition, we say $\{T_1, T_2, \ldots, T_k\}$ *is strongly compact-double-colliding if* $\{T_1, \ldots, T_k\} \subseteq \Sigma^{\mathsf{min_match}+1}$.

While it is generally easier to construct collision-based amplification queries from a strongly compact-double-colliding set, such a set of suitable size may not exist for some DEFLATE hash functions, especially if the hash function is not rolling. We do not give constructions of attacks that require strongly double-colliding sets in this paper.

A weaker variant of double-colliding, *weakly double-colliding*, may be of interest if the hash function has some unusual properties that make finding double-colliding sets hard, $\mathsf{hash_bits}$ is large, or the alphabet Σ is small.

Definition 8 (Weakly double-colliding set). *Let* k, $\mathsf{min_match}$, $\mathsf{hash_bits} \in \mathbb{N}_+$, *where* $\mathsf{min_match} \geq 2$, *let* $\Sigma \subseteq \{0, \ldots, 255\}$, $v_1, v_2 \in \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\}$, *and let* H *be a DEFLATE hash function with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$. *A set of* k *byte strings* $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^{>\mathsf{min_match}}$ *is weakly double-colliding at* $\{v_1, v_2\}$ *with regard to* H *on* Σ, *if for every* $i \in [k]$, *it holds that*

$$\{\mathsf{H}(T_i\,[1 : \mathsf{min_match}]), \mathsf{H}(T_i\,[2 : \mathsf{min_match} + 1])\} = \{v_1, v_2\}\,.$$

In addition, we say $\{T_1, T_2, \ldots, T_k\}$ *is weakly compact-double-colliding if* $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^{\mathsf{min_match}+1}$.

It is easy to see that a strongly double-colliding set at v with regard to H is also double-colliding at v with regard to H, and a double-colliding set at v with regard to H is also weakly double-colliding at $\{v, v\} = \{v\}$ with regard to H.

We do not explicitly give constructions of attacks based on weakly double-colliding sets, as we can find double-colliding sets of alphanumeric strings suitable for our constructions for all DEFLATE implementations we examined. Nevertheless, we will briefly describe how to modify our collision-based amplification technique when only weakly double-colliding sets are available.

If Σ is too small or $\mathsf{hash_bits}$ is too large, then we may not even be able to find any weakly double-colliding set on Σ of reasonable size, making collision-based amplification much harder. We do not discuss this case as we do not expect it to occur frequently in practice, but we note that one can further weaken Definition 8 by considering more than two hash values v_1, v_2 for collisions.

Finally, as we will see later, it is of special interest to consider colliding or double-colliding sets of strings that are *diverse*, i.e. each string starts with a distinct byte.

Definition 9 (Diverse set). *Let* $k \in \mathbb{N}_+$. *A set of* k *strings* $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^+$ *is diverse, if* $k > 1$, *and for every* $i, j \in [k]$, *it holds that* $T_i[1] = T_j[1] \implies i = j$.

We remark that, from a diverse colliding (resp. diverse double-colliding) set, one can construct a compact-colliding (resp. compact-double-colliding) set of the same size by truncating each string in the diverse set, but the reverse direction does not necessarily hold.

Apart from zlib-ng, all DEFLATE implementations we examined have diverse double-colliding sets of at least three alphanumeric strings. However, there are no diverse double-colliding sets of ASCII-printable strings for zlib-ng.

7.2 Step-by-step introduction

As with our general amplification techniques in Section 6, the main goal of collision-based amplification is to craft queries that increase the compressed length difference to outweigh noise in the compression side channel. At the same time, because of the upper bound L_Q on the query length in our attack model, it is of interest to design compact queries for amplification, i.e. create the same level of compressed length difference using queries as short as possible.

[81] Note that we do not require $c^{\mathsf{min_match}} \in \{T_1, T_2, \ldots, T_k\}$.

We now present the ideas behind collision-based amplification. As this technique is more intricate compared to our amplification techniques in Section 6, we will show the core ideas of collision-based amplification and motivate some of our design tricks by first sketching a basic construction and then making several improvements to it, focussing on concrete examples rather than formal description and analysis.

Our introduction assumes that the DEFLATE compressor uses the deflate_slow compression strategy, so that LZ77 inserts every position in the sliding window into the hash table and performs lazy matching. All of our constructions are based on the general framework of chaining in Section 6.3.

Setup. To facilitate our discussion, we fix the configuration of the zlib compressor in our example setting. We set the compression level of zlib to 4 and other parameters to default. The compression strategy at compression level 4 in zlib is deflate_slow, and we have min_match = 3, max_match = 258, hash_bits = 15, good_length = 4, max_lazy = 4, nice_length = 16, and max_chain = 16.[82]

Recall in Section 3.4 that, at each new position, LZ77 in zlib computes the hash value of the substring at the current position, and checks the corresponding chain in the hash table for a match. The search stops before the entire chain has been traversed if a match of at least nice_length bytes is found or max_chain entries have been checked. In deflate_slow, if the match found by LZ77 is strictly shorter than max_lazy bytes, then LZ77 performs lazy matching by looking for longer matches at the next position, where LZ77 checks at most $\lfloor \mathsf{max_chain}/4 \rfloor$ entries in the hash table if the previous match is at least good_length bytes.

For the zlib hash function with default parameters, the maximum size of a compact-colliding set of alphanumeric strings is 27; we give such an example in Table 3.

`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 3: The 27 alphanumeric strings with the hash value 20081 when using the default zlib hash function.

Because the zlib hash function is rolling and the string `QQQ` is in the compact-colliding set represented in Table 3, by Theorem 6, we can obtain a compact double-colliding set of size 27 by appending the byte `Q` to each of the strings in Table 3. The resulting set is also strongly double-colliding as it contains `QQQQ`.

7.2.1 Basic collision-based chaining

Construction. As a first attempt, we introduce a basic construction for collision-based chaining. Assume the attacker knows a set of m strings $\{T_1, \ldots, T_m\}$ that is double-colliding at v on Σ with regard to the DEFLATE hash function used by the compressor, where m is even, $\mathsf{max_chain}/4 < m \leq \mathsf{max_chain}/2 - 2$,[83] and $\xi := \sum_{i=1}^{m} |T_i| < \mathsf{max_match}$. For the parameters in our example setting, this means $m = 6$. Note that such a set cannot exist under some DEFLATE parameter choices, such as when $(\lfloor \mathsf{max_chain}/4 \rfloor + 1) \cdot (\mathsf{min_match} + 1) \geq \mathsf{max_match}$.

The query Q follows the query structure for the chaining techniques described in Section 6.3, i.e.

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

Here dict_1 represents only one byte string $g\|T_1\|T_2\|\cdots\|T_m$, filler is the empty string ε, dict_2 represents the sequence of byte strings $(T_1, \ldots, T_{m-1}, T_m\|R)$, where $R \in \Sigma$, and $S = T_1\|T_2\|\cdots\|T_m\|R$. The purpose of the one-byte "glue" string R is to preserve the difference in LZ77 pointer positions depending on whether $ck[1] = g$; that is, LZ77 likely moves the pointer to R if and only if $ck[1] \neq g$. As we will see, the use of R enables us to extend the chain, and, later in Section 8, use collision-based amplification as a component of more complex queries. In this construction, R can be set almost arbitrarily.

[82] As we will see in Section 7.4, one can also construct collision-based amplification queries against zlib level 4 using colliding sets instead of double-colliding sets, because max_lazy is very small.

[83] This assumption implies $m > 2$.

In our example setting, a suitable double-colliding set is $\{\texttt{0pqQ}, \texttt{1PqQ}, \texttt{20qQ}, \texttt{Pr1Q}, \texttt{R21Q}, \texttt{qR1Q}\}$ with $m = 6$. Setting $R = \texttt{a}$, the query for $g = \texttt{1}$ can be

```
-10pqQ1PqQ20qQPr1QR21QqR1Q--0pqQ-1PqQ-20qQ-Pr1Q-R21Q-qR1Qa-
secret=10pqQ1PqQ20qQPr1QR21QqR1Qa
```

Here and in the rest of Section 7, we use - as the separator in dict_1 and dict_2 in our example queries, and we split each example query into multiple lines for better readability.

We illustrate the behaviour of LZ77 in our example in Fig. 3.

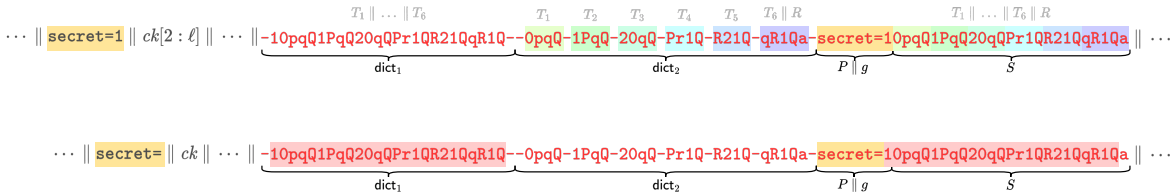


Fig. 3: Illustration of basic collision-based chaining. The query Q is in red and the likely LZ77 matches are highlighted. It shows the effect of amplification constructed using collisions if the guess for $ck[1]$ is correct (top) and when it is not (bottom).

If $ck[1] = g$, then LZ77 likely encodes $P\|g\|S$ in the query Q as

$$\texttt{<L}(\ell_P + 1)\texttt{,D*>}\|\texttt{<L}|T_1|\texttt{,D*>}\|\cdots\|\texttt{<L}|T_{m-1}|\texttt{,D*>}\|\texttt{<L}(|T_m| + 1)\texttt{,D*>},$$

where the back-references correspond to $P\|g, T_1, \ldots, T_{m-1}, T_m\|R$, which are `secret=1`, `0pqQ`, `1PqQ`, `20qQ`, `Pr1Q`, `R21Q`, `qR1Qa` in our example. All back-references except the first one are expected to point to a byte string in dict_2.

Otherwise, if $ck[1] \neq g$, then LZ77 likely encodes $P\|g\|S$ in the query Q as

$$\texttt{<L}\ell_P\texttt{,D*>}\|\texttt{<L}\ell_S\texttt{,D*>}\|R,$$

where the back-references and literals correspond to P, $g\|T_1\|\cdots\|T_m$ and R, which are `secret=`, `10pqQ1PqQ20qQPr1QR21QqR1Q` and `a` in our example. The second back-reference is expected to point to a byte string in dict_1.

Analysis. We informally justify the output of LZ77 for $P\|g\|S$ in the query. We briefly remark the inaccuracies in our analysis in footnotes, which can be ignored by readers for the purpose of understanding the general idea. We assume that LZ77 finds a match for $P\|g$ in the sliding window and moves the pointer to $S[1]$ in the query Q if $ck[1] = g$, and LZ77 finds a match for P in the sliding window and moves the pointer to g in the query Q otherwise.

- If $ck[1] = g$, then by assumption, the LZ77 pointer is now at the first byte of T_1 in S. Next, LZ77 looks for a match at the current position by checking the entries of the hash chain corresponding to $T_1[1 : \mathsf{min_match}]$, whose hash value is v by definition. After checking about $2m < \mathsf{max_chain}$ entries of the hash chain,[84] two from each byte string in dict_2,[85] LZ77 likely finds a back-reference to T_1 in dict_2. However, LZ77 likely cannot find a longer match, because the nearest match that is longer than T_1 is expected to be $T_1\|\cdots\|T_m$ in dict_1,[86] which cannot be found by LZ77 as it requires checking $4m > \mathsf{max_chain}$ entries of the same hash chain. Therefore, LZ77 likely finds the match T_1 in dict_2 at the pointer position $S[1]$.
  As the result of using a double-colliding set, the match T_1 found by LZ77 likely cannot be improved by lazy matching. In more detail, if $|T_1| < \mathsf{max_lazy}$, then LZ77 looks for a match at the next position

[84] The exact number of entries to check may be slightly larger than $2m$; for example, some substring of P may also have the hash value v.

[85] A byte string $T \in \{T_i\}_{i\in[m]}$ may contain more than two $\mathsf{min_match}$-length substrings with the hash value v.

[86] This actually depends on the composition of $\{T_i\}_{i\in[m]}$; for example, if $T_1\|T_2[1]$ is a substring of T_m, then the nearest match longer than $|T_1|$ is closer than expected.

in the hash chain corresponding to $T_1\,[2 : \mathsf{min_match} + 1]$, whose hash value is also v by definition. The nearest match that is longer than the current match T_1 is expected to be $T_1[2 : |T_1|]\|\cdots\|T_m$ in dict_1,[87] which cannot be found by LZ77 as it requires checking $4m > \mathsf{max_chain}$ entries of the same hash chain. Otherwise, if $|T_1| \geq \mathsf{max_lazy}$, then LZ77 does not perform lazy matching.

In summary, at the pointer position $S[1]$, the first byte of T_1, LZ77 likely outputs a back-reference to T_1 in dict_2 and moves the pointer to $S[|T_1| + 1]$, the first byte of T_2. Continuing this process, we conclude that LZ77 likely finds matches for $T_1, \ldots, T_{m-1}, T_m\|R$ in dict_2 and moves the pointer just past S.

– Otherwise, if $ck[1] \neq g$, then by assumption, the LZ77 pointer is at g just before the chain S. Next, LZ77 looks for a match by checking the hash chain corresponding to $(g\|S)\,[1 : \mathsf{min_match}]$, whose hash value is expected to be different from v.[88] LZ77 finds $g\|T_1\|T_2\|\cdots\|T_m$ in dict_1, which cannot be extended or improved by lazy matching.[89] Therefore, LZ77 likely outputs a back-reference to $g\|T_1\|T_2\|\cdots\|T_m$ in dict_1 and moves the pointer to R, the last byte of S.

Performance. If $ck[1] = g$, then the LZ77 output for $P\|g\|S$ likely consists of $m + 1$ back-references, most of which point to dict_2; otherwise, the LZ77 output likely only consists of two back-references and one literal. 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 should be proportional to m. However, when $ck[1] = g$, the back-references have shorter lengths and distances and tend to require fewer bytes to encode, so the actual compressed length difference may be smaller, especially if m is small. Note that m is upper-bounded by $\min\{\mathsf{max_chain}/2 - 1, \mathsf{max_match}/(\mathsf{min_match} + 1)\}$, which is typically quite small (7 in our example). The length of the query depends on the choice of the double-colliding set $\{T_i\}_{i\in[m]}$; it is about $(3 \cdot \mathsf{min_match} + 4)m$ bytes if the set is compact-double-colliding.

7.2.2 Extending the chain

The main drawback of our basic collision-based chaining construction is that it cannot achieve a large compressed length difference, because both $\mathsf{max_chain}$ and $\mathsf{max_match}$ are typically quite small. In our example setting, $\mathsf{max_chain} = 16$, and $\mathsf{max_match} = 258$, so m cannot exceed 6, and the compressed length difference can hardly be longer than a few bytes.

A straightforward improvement is to find another set of strings $\{T'_i\}_{i\in[m']}$ that is double-colliding at some hash value $v' \neq v$ on Σ, use this set to construct another collision-based chaining query, and piece together this query with the existing query. Here we similarly require that m' is even, $\mathsf{max_chain}/4 < m' \leq \mathsf{max_chain}/2 - 2$, and $\xi' := \sum_{i=1}^{m'} |T'_i| < \mathsf{max_match}$.

Construction. More formally, for the query $Q = \mathsf{dict}_1\|\mathsf{filler}\|\mathsf{dict}_2\|P\|g\|S$, dict_1 now represents the sequence of two byte strings $(g\|T_1\|T_2\|\cdots\|T_m, R\|T'_1\|T'_2\|\cdots\|T'_{m'})$, filler is the empty string ε, dict_2 now represents the sequence of byte strings $(T_1, \ldots, T_{m-1}, T_m\|R, T'_1, \ldots, T'_{m'-1}, T'_{m'}\|R')$, and

$$S = T_1\|\cdots\|T_m\|R\|T'_1\|\cdots\|T'_{m'}\|R',$$

where $R, R' \in \Sigma$. As before, the purpose of R' is to preserve the difference in LZ77 pointer positions, and, informally speaking, the purpose of R is to "glue" the two chains.

In our example setting, using another set $\{\mathtt{pPS3}, \mathtt{qs33}, \mathtt{s1s3}, \mathtt{PQs3}, \mathtt{1pS3}, \mathtt{30S3}\}$ that is double-colliding at $v' = 19027$ with $m' = 6$, and setting $R = R' = \mathtt{a}$, one can construct a longer query for $g = \mathtt{1}$ as

```
-10pqQ1PqQ20qQPr1QR21QqR1Q-apPS3qs33s1s3PQs31pS330S3-
-0pqQ-1PqQ-20qQ-Pr1Q-R21Q-qR1Qa-pPS3-qs33-s1s3-PQs3-1pS3-30S3a-
secret=10pqQ1PqQ20qQPr1QR21QqR1QapPS3qs33s1s3PQs31pS330S3a
```

[87] This also depends on other factors, like the prefix P and the composition of $\{T_i\}_{i\in[m]}$.

[88] Note that the validity of this statement depends on the choice of the guess g, which cannot be chosen arbitrarily, and in theory the statement might not hold for certain choices of g under highly atypical compressor configurations without changing $\{T_i\}_{i\in[m]}$.

[89] Formally speaking, this statement is only true under appropriate assumptions, which are not given here. We can enforce this statement by requiring that $g\|T_1\|T_2\|\cdots\|T_m\|R$ do not appear in the sliding window before g in S (note that lazy matching is never performed because the match is longer than $\mathsf{max_lazy} = 4$ bytes).

Performance. If $ck[1] = g$, then the LZ77 output for $P\|g\|S$ likely consists of $m+m'+1$ back-references, which correspond to $P\|g, T_1, \ldots, T_{m-1}, T_m\|R, T'_1, \ldots, T'_{m'-1}, T'_{m'}\|R'$. All except the first back-reference are expected to point to dict_2. Otherwise, the LZ77 output likely only consists of three back-references, which correspond to $P, g\|T_1\|T_2\|\cdots\|T_m, R\|T'_1\|T'_2\|\cdots\|T'_{m'}$, as well as one literal R'. The last two back-references are expected to point to dict_1, The compressed length difference is expected to be proportional to $m + m'$.

It is easy to see that one can further extend the query in the same manner using double-colliding sets for different hash values. However, for some DEFLATE hash functions, there may only be a limited number of hash values at which one can find suitable double-colliding sets. In addition, the query is more likely to fail as it grows longer and uses more double-colliding sets, and the length of the query is still bounded by the sliding window size $\mathsf{window_size}$.

7.2.3 A more efficient extension

It is also possible to construct a query that creates a larger compressed length difference without requiring another double-colliding set. A benefit of using the same double-colliding set is that shorter queries are sufficient to achieve the same compressed length difference.

As a first attempt toward this goal, one might set $m' = m$ and $T'_i = T_i$ for each $i \in [m]$ in the above construction. We briefly discuss why this would not work as desired. Assume for simplicity that $R = R'$. If $ck[1] = g$, then the LZ77 pointer is expected to move to $T'_1[1] = T_1[1]$ in S at some point, where the desired behaviour for LZ77 is to output a back-reference for T'_1 and move the pointer to $T'_2[1]$. However, LZ77 likely finds a match $T_1\|\cdots\|T_m\|R = T'_1\|\cdots\|T'_m\|R'$ at the start of S after checking about $2m < \mathsf{max_chain}$ entries in the hash chain for v. Therefore, the LZ77 output for $P\|g\|S$ likely only consists of $m + 2$ back-references instead of $2m + 1$, rendering the extension ineffective.

Construction. Our solution to this problem is to slightly change the order of the strings when placing them for the second time in the chain S. To simplify our discussion, we additionally assume the set $\{T_i\}_{i\in[m]}$ is diverse double-colliding, which is true for our example set, but a weaker property also suffices. We also require that $\xi = \sum_{i=1}^{m}|T_i| \le \mathsf{max_match} - 2$. The query Q uses the same structure $\mathsf{dict}_1\|\mathsf{filler}\|\mathsf{dict}_2\|P\|g\|S$, where dict_1 represents the sequence of two byte strings

$$(g\|T_1\|T_2\|\cdots\|T_{m-1}\|R'\|T_m,\ R\|T_2\|T_1\|\cdots\|T_m\|R\|T_{m-1})\,,$$

filler is the empty string ε, dict_2 represents a single byte string $T_{m-1}\|R'$, and the chain S is

$$T_1\|T_2\|\cdots\|T_{m-1}\|R'\|T_m\|R\|T_2\|T_1\|\cdots\|T_m\|R\|T_{m-1}\|R',$$

where $R, R' \in \Sigma$. Note that $T_1, \ldots, T_m$ is permuted in the second parts of dict_1 and S.

This construction is more efficient than the previous one, because it only adds one byte string instead of $2m$ byte strings to dict_2. The reason we can use a much shorter dict_2 in this construction is that the second part of dict_1 essentially acts as dict_2 for the first half of S, which in turn essentially acts as dict_2 for the second half of S. The main purpose of dict_2 in this construction is to preserve the difference in LZ77 pointer positions.[90]

In our example setting, using $R = \mathtt{a}$ and $R' = \mathtt{b}$, one can construct an extended query for $g = \mathtt{1}$ as

```
-10pqQ1PqQ20qQPr1QR21QbqR1Q-a1PqQ0pqQPr1Q20qQqR1QaR21Q--R21Qb-
secret=10pqQ1PqQ20qQPr1QR21QbqR1Qa1PqQ0pqQPr1Q20qQqR1QaR21Qb
```

The query is now much shorter compared to the query using two double-colliding sets, while it is expected to create a similar compressed length difference.

Performance. If $ck[1] = g$, then LZ77 likely encodes $P\|g\|S$ in the query Q as $2m + 1$ back-references, which correspond to $P\|g, T_1, T_2, \ldots, T_{m-2}, T_{m-1}\|R', T_m\|R, T_2, T_1, \ldots, T_{m-3}, T_m\|R, T_{m-1}\|R'$, where the first half of the back-references generally point to dict_1, and the second half of them generally point to S. Otherwise, if $ck[1] \neq g$, then LZ77 likely encodes $P\|g\|S$ as 3 back-references, which correspond to P, $g\|T_1\|T_2\|\cdots\|T_{m-1}\|R'\|T_m$ and $R\|T_2\|T_1\|\cdots\|T_m\|R\|T_{m-1}$, as well as one literal R'. The last two back-references are expected to point to dict_1. The compressed length difference is expected to be proportional to $2m$.

[90] In more detail, if we remove dict_2, then we need to append R' to the byte string $R\|T_2\|T_1\|\cdots\|T_m\|R\|T_{m-1}$ in dict_1 to enable LZ77 to find a back-reference for $T_{m-1}\|R'$ in the first half of S. However, after this modification, LZ77 will likely move the pointer to R', the last byte of S, irrespective of whether $ck[1] = g$.

Analysis. We can informally justify the LZ77 output when $ck[1] = g$ by repeating the following arguments. Suppose the LZ77 pointer is at the first byte of the first occurrence of T_i in S for some $i \in [m-2]$. LZ77 looks for a match on the hash chain for v and likely finds T_i in the second part of dict_1 after checking about $2m+4$ entries if i is even, or about $2m$ entries if i is odd. Note that T_i is followed by T_{i+1} in S and the match found in dict_1 is followed by either T_{i-1} or T_{i+3}, so because $\{T_i\}_{i\in[m]}$ is diverse, the match found at that position cannot be longer than T_i. LZ77 also cannot find $T_i\|\cdots\|T_{m-1}\|R'\|T_m$ in the first part of dict_1, because that would require checking at least $4m+2 > \mathsf{max_chain}$ entries. The LZ77 output when $ck[1] \neq g$ follows similarly from the analysis for the extension using two double-colliding sets.

One can clearly further extend the query using different permutations of $T_1, \ldots, T_m$ that satisfy certain conditions. However, for each new permutation used in the query, one needs to also add its corresponding entries to the dictionaries, and the query length is still limited by the sliding window size.

7.2.4 Basic bootstrapping

For the last set of our example constructions, we show how to extend the query constructed using a single double-colliding set as described above via *bootstrapping*, without adding more entries to the dictionaries. Our bootstrapping construction is based on a simple observation: the above construction can still work as intended with minor modifications if we repeat the content of S multiple times, because a previous occurrence of S can substitute for the functionalities of dict_1 and dict_2 for its subsequent occurrences in the query. This enables us to obtain a compressed length difference that is linear in the number of repetitions of S.

Construction. Let $n \in \mathbb{N}_+$, such that $2(n-1)\cdot(2+\xi) \leq \mathsf{max_match}$, where $\xi = \sum_{i=1}^{m}|T_i|$. Our basic bootstrapping construction is the same as the above construction, except that $R' = g$, and the chain S is $(S')^n$, where S' is the chain in the above construction; that is, the chain S is

$$(T_1\|T_2\|\cdots\|T_{m-1}\|R'\|T_m\|R\|T_2\|T_1\|\cdots\|T_m\|R\|T_{m-1}\|R')^n\,.$$

In our example setting, one can construct a longer query for $g = \mathtt{1}$ as

```
-10pqQ1PqQ20qQPr1QR21Q1qR1Q-a1PqQ0pqQPr1Q20qQqR1QaR21Q--R21Q1-
secret=10pqQ1PqQ20qQPr1QR21Q1qR1Qa1PqQ0pqQPr1Q20qQqR1QaR21Q1
0pqQ1PqQ20qQPr1QR21Q1qR1Qa1PqQ0pqQPr1Q20qQqR1QaR21Q1
```

with $n = 2$, $R = \mathtt{a}$ and $R' = \mathtt{1}$.

We illustrate the behaviour of LZ77 in our example in Fig. 4.



Fig. 4: Illustration of basic bootstrapping in the example setting. The query Q is in red and the likely LZ77 matches are highlighted. Hatched background signifies the final longer matches; the part of the query with pink background will be replaced by a back-reference to $S[0 : 2(n-1)\cdot(2+\xi)-1] = S[0 : (n-1)|S'|-1]$.

Performance. If $ck[1] = g$, then LZ77 likely encodes $P\|g\|S$ in the query Q as $2mn+1$ back-references, with most of them pointing to previous positions in S. Otherwise, if $ck[1] \neq g$, then LZ77 likely encodes $P\|g\|S$ as 4 back-references, the last of which corresponds to a very long match of length $2(n-1)\cdot(2+\xi)$ in S, namely $(R'\|T_1\|T_2\|\cdots\|T_{m-1}\|R'\|T_m\|R\|T_2\|T_1\|\cdots\|T_m\|R\|T_{m-1})^{(n-1)}$,[91] as well as one literal R'. The compressed length difference is expected to be proportional to $2mn$.

However, note that our construction requires $n \leq \mathsf{max_match}/(4+2\xi)+1$, which implies $n \leq 5$ in our example setting. Accordingly, the chain S cannot be longer than $(n/(n-1))\cdot\mathsf{max_match}$, much smaller than $\mathsf{window_size}$ for typical compressor configurations. We now informally explain this restriction. When $ck[1] \neq g$, LZ77 likely first finds three back-references, which correspond to the prefix P and the two entries of dict_1, and moves the pointer to $S[|S'|] = S[4+2\xi]$. If n is larger than required, then at the pointer position $S[4+2\xi]$, instead of finding a back-reference that corresponds to the very long match stated above, LZ77 likely finds a back-reference of length $\mathsf{max_match}$, a prefix of the desired match, and moves the pointer to $S[4+2\xi+\mathsf{max_match}]$, after which the behaviour of LZ77 is out of our control and likely converges to the behaviour of LZ77 when $ck[1] = g$ at some point. The restriction on n limits the compressed length difference resulting from this construction, especially if $\mathsf{max_match}$ is small.

7.2.5 Advanced bootstrapping

Finally, we sketch several methods for removing the restriction on n in our bootstrapping construction, thereby removing any non-trivial limit on the compressed length difference. Because these methods are too complex to describe formally as a part of our introduction, here we only present their core ideas and apply them to our example query for the bootstrapping construction.

Aligning segments. One can choose $\{T_i\}_{i\in[m]}$ such that $2+\xi$ divides $\mathsf{max_match}$, where $\xi = \sum_{i=1}^{m}|T_i|$. This way, when $ck[1] \neq g$, the LZ77 pointer likely always falls right in the middle or at the last byte of a segment S' in the chain S. To construct such a set, one can append some bytes to each byte string in an existing double-colliding set until the new set meets the requirement on its total length.

In our example setting, one can use $\{\mathtt{0pqQABC}, \mathtt{1PqQDEF}, \mathtt{20qQGHI}, \mathtt{Pr1QJKL}, \mathtt{R21QLMN}, \mathtt{qR1QYZ}\}$ with $m = 6$. We now have $2+\xi = 43$, which divides $\mathsf{max_match} = 258$. The query for $g = \mathtt{1}$ can be

```
-10pqQABC1PqQDEF20qQGHIPr1QJKLR21QLMN1qR1QYZ-
a1PqQDEF0pqQABCPr1QJKL20qQGHIqR1QYZaR21QLMN--R21QLMN1-secret=1
0pqQABC1PqQDEF20qQGHIPr1QJKLR21QLMN1qR1QYZa1PqQDEF0pqQABCPr1QJKL20qQGHIqR1QYZaR21QLMN1
0pqQABC1PqQDEF20qQGHIPr1QJKLR21QLMN1qR1QYZa1PqQDEF0pqQABCPr1QJKL20qQGHIqR1QYZaR21QLMN1
```

with $n = 2$, $R = \mathtt{a}$ and $R' = \mathtt{1}$. Here n can be arbitrarily large. In particular, n is not limited by $\mathsf{window_size}$. However, the query preserves the difference in pointer positions if and only if $n\cdot\mathsf{max_match}$ divides $(n-1)|S|$, which means $n-1$ should be a multiple of 3 in our example.[92] If $ck[1] = g$, then LZ77 likely encodes $P\|g\|S$ in the query Q as $2mn+1$ back-references. Otherwise, if $ck[1] \neq g$, then LZ77 likely encodes $P\|g\|S$ as $3+\lceil(n-1)|S|/(n\cdot\mathsf{max_match})\rceil$ back-references, and, if $n\cdot\mathsf{max_match}$ divides $(n-1)|S|$, one literal R'. Because the difference in the LZ77 output size grows linearly to n, the compressed length difference can be as large as possible, only limited by the maximum query length.

Optimisation. It is possible to achieve a similar effect as the previous method more efficiently by slightly modifying our bootstrapping construction. Here we show one such case in our example setting. One can use the double-colliding set $\{\mathtt{0pqQA}, \mathtt{1PqQ}, \mathtt{20qQ}, \mathtt{Pr1Q}\}$ of $m = 4$ elements[93] to construct the query for $g = \mathtt{1}$ as

```
-10pqQA0pqQA1PqQ20qQ1Pr1Q-a1PqQ0pqQAPr1Qa20qQ--20qQ1-
secret=10pqQA0pqQA1PqQ20qQ1Pr1Qa1PqQ0pqQAPr1Qa20qQ1
0pqQA0pqQA1PqQ20qQ1Pr1Qa1PqQ0pqQAPr1Qa20qQ1
```

[91] The match is possible because LZ77 in DEFLATE is self-referencing; that is, a back-reference can point to a previous substring that overlaps the matched substring.

[92] Whether this condition is met has almost no impact on the compressed length difference. However, this condition is required if we later use this method to construct an amplify gadget (Section 8.4).

[93] Although we require $m > \mathsf{max_chain}/4 = 4$ for the original construction, our modified construction introduces more collisions in each segment and therefore allows $m = \mathsf{max_chain}/4 = 4$.

with $n = 2$, $R = \texttt{a}$ and $R' = \texttt{1}$. Again, n can be arbitrarily large and is not limited by window_size, because the length of a segment is $|S|/n = 43$, a divisor of max_match $= 258$, but the query preserves the difference in pointer positions if and only if $n \cdot$ max_match divides $(n-1)|S|$, which means $n-1$ should be a multiple of 6 in our example. In order to build a segment of length 43, an odd length, we construct the two entries of dict_1 such that their lengths are of different parities, which we achieve by repeating `0pqQA`, a byte string of length 5 in the double-colliding set, within the first entry of dict_1. As before, if $ck[1] = g$, then LZ77 likely encodes $P\|g\|S$ in the query Q as $2mn + 1$ back-references. Otherwise, if $ck[1] \neq g$, then LZ77 likely encodes $P\|g\|S$ as $3 + \lceil (n-1)|S|/(n \cdot \text{max_match}) \rceil$ back-references, and, if $n \cdot$ max_match divides $(n-1)|S|$, one literal R'.

Aligning collisions. Alternatively, one can choose $\{T_i\}_{i \in [m]}$ to ensure that the LZ77 pointer never falls at a position within S of hash value v when $ck[1] \neq g$. Intuitively, this ensures that the behaviour of LZ77 is not affected by collisions at v when $ck[1] \neq g$. The most straightforward way to achieve this goal is to ensure that the distance between the first bytes of any two neighbouring T_i, T_j in the chain S is the same divisor of max_match, so that the LZ77 pointer is expected to always fall just before some T_i in the chain S when $ck[1] \neq g$, where LZ77 likely finds a match from a previous position in S or dict_1 because the hash value at this position is likely different from v.

In our example setting, using the double-colliding set {`0pqQAB`, `1PqQCD`, `20qQEF`, `Pr1QGH`, `R21QI`, `qR1QJ`} with $m = 6$ elements, one can construct the query for $g = \texttt{1}$ as

```
-10pqQAB1PqQCD20qQEFPr1QGHR21QI1qR1QJ-a1PqQCD0pqQABPr1QGH20qQEFqR1QJaR21QI--R21QI1-
secret=10pqQAB1PqQCD20qQEFPr1QGHR21QI1qR1QJa1PqQCD0pqQABPr1QGH20qQEFqR1QJaR21QI1
0pqQAB1PqQCD20qQEFPr1QGHR21QI1qR1QJa1PqQCD0pqQABPr1QGH20qQEFqR1QJaR21QI1
```

with $n = 2$, $R = \texttt{a}$ and $R' = \texttt{1}$. Here n can be arbitrarily large and is not limited by window_size. However, as before, the query preserves the difference in pointer positions if and only if $n \cdot$ max_match divides $(n-1)|S|$, which means $n-1$ should be a multiple of 43 in our example.[94] The distance between any neighbouring collisions in the chain is 6, which divides max_chain $= 258$. If $ck[1] = g$, then LZ77 likely encodes $P\|g\|S$ in the query Q as $2mn + 1$ back-references, and most of them are `<L6,D*>` in our example. Otherwise, if $ck[1] \neq g$, then LZ77 likely encodes $P\|g\|S$ as $3 + \lceil (n-1)|S|/(n \cdot \text{max_match}) \rceil$ back-references, and, if $n \cdot$ max_match divides $(n-1)|S|$, one literal R'; most of the back-references would be `<L258,D72>` in our example if we used a large n.

Concrete performance. To give an idea of the performance of advanced bootstrapping, we measure the performance of the three example constructions shown above in our example setting. For simplicity, we fix the zlib compressor input as `secret=`$\|ck\|Q$ in the compression length oracle, and we assume that the cookie ck is either `0` or `1`. The results are shown in Table 4. The constructions work as desired even with queries of size $\ell_Q \approx 100$MiB, creating compressed length differences greater than $\ell_Q/10$ in all cases. However, due to Huffman coding, longer queries are slightly less efficient (i.e. the ratio between the compressed length difference and the query length is smaller for longer queries). We will evaluate the performance of collision-based amplification more rigorously in Section 9.

7.2.6 Automating the technique

Constructing collision-based amplification queries as we have described them so far is still a complex task. First, our example constructions only work for a subset of DEFLATE compressor configurations; for example, in zlib, only compression levels 4, 5 and 6 satisfy our requirements. Second, these constructions rely on some unspecified assumptions on the double-colliding set, so they may produce queries that do not work as desired even when other assumptions are met. Finally, the most powerful form of collision-based amplification requires the advanced bootstrapping technique, for which we did not provide a detailed description. As a result, some manual effort is necessary, which can become time-consuming if the double-colliding set is very large.

In order to remove the need for human intervention as much as possible from the process of building collision-based amplification queries, we introduce the concept of *templates*. A template is a deterministic algorithm that takes as input the prefix P and the guess g, some parameters and an *instance set*, which is a set of byte strings that satisfy some special properties, and outputs a collision-based amplification query.

[94] Similarly, this condition is only required when we construct an amplify gadget (Section 8.4) using this method.

	$\ell_Q \approx$ 100KiB		$\ell_Q \approx$ 100MiB	
	$ck = \texttt{0}$	$ck = \texttt{1}$	$ck = \texttt{0}$	$ck = \texttt{1}$
Aligning segments	461B	11.7KiB	348KiB	10.8MiB
Optimisation	387B	15.1KiB	298KiB	14.8MiB
Aligning collisions	452B	12.5KiB	348KiB	11.5MiB

Table 4: Performance of the three advanced bootstrapping constructions in our example setting. For each construction, we choose the largest n such that the query size ℓ_Q does not exceed 100KiB (resp. 100MiB); note that the resulting queries do not necessarily preserve the difference in pointer positions. The column $ck = \texttt{0}$ (resp. $ck = \texttt{1}$) shows the compressed lengths when $ck = \texttt{0}$ (resp. $ck = \texttt{1}$), rounded to three significant figures.

By using templates, the problem of building a collision-based amplification query is reduced to finding a suitable instance set, which can be automated, or, for more efficient queries, semi-automated. We have implicitly presented several templates in Sections 7.2.1 to 7.2.5. In Sections 7.3 and 7.4, we describe and analyse some more templates, each intended for a different setting. Together they cover most DEFLATE implementations and configurations. These templates are similar to our example constructions, but they are much more versatile and precise.

We note that there are many possible ways to construct collision-based amplification queries besides using our templates; in fact, for any specific compressor implementation and configuration, it is usually possible to manually construct queries that are more efficient than the queries built using our templates.

7.3 Templates for deflate_slow

We first consider deflate_slow, the default compression strategy for most DEFLATE implementations. We discuss three possible cases for the size of the largest diverse double-colliding set known by the attacker, each using a different template, as well as the case where no double-colliding sets exist. For simplicity, we use LZ77 to denote the LZ77 algorithm with respect to some (unspecified) DEFLATE implementation and configuration using the deflate_slow strategy.

7.3.1 deflate_slow instance set

We give the definition of a deflate_slow instance set, which we use to instantiate our templates.

Definition 10. *(*deflate_slow *instance set) Let* $k, m, \mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$, $\Sigma \subseteq \{0, \ldots, 255\}$, *and let* H *be a DEFLATE hash function with the parameters* min_match *and* hash_bits. *A set of* km *byte strings* $\mathcal{T} = \{T_{1,1}, T_{1,2}, \ldots, T_{1,m}, \ldots, T_{k,1}, T_{k,2}, \ldots, T_{k,m}\} \subseteq \Sigma^*$ *is a* (k, m)-deflate_slow *instance set with regard to* H *on* Σ, *if*

1. *For every* $(i_1, \ldots, i_m) \in [k]^m$, $\{T_{i_1,1}, \ldots, T_{i_m,m}\}$ *is diverse;*
2. *For every* $j \in [m]$, $\{T_{1,j}, \ldots, T_{k,j}\}$ *is double-colliding at some hash value* v_j*;*
3. *For every* $T_1, T_2, T_3, T_4 \in \mathcal{T}$, *where* $T_1 \neq T_3$ *or* $T_2[1] \neq T_4[1]$, *it holds that* $T_1\,[2 : |T_1|]\,\|T_2[1]$ *is not a substring of* $T_3 \| T_4$.[95]

In addition, we say that $\mathcal{T}$ *is a consistent* (k, m)-deflate_slow *instance set if* $\mathcal{T}$ *is double-colliding at some hash value* v, *i.e.* $v_j = v$ *for every* $j \in [m]$.

For example, {0pqQAAA, 1PqQAAA, 20qQAAA, 0pqQBBB, 1PqQBBB, 20qQBBB} is a (2,3)-deflate_slow instance set with regard to the default hash function in zlib. Note that, for each $j \in [m]$, it is not required that $T_{1,j}, \ldots, T_{k,j}$ start with the same byte, even though it should be more convenient to construct $T_{1,j}, \ldots, T_{k,j}$ by appending different suffixes to the same byte string.

Finding a consistent (k, m)-deflate_slow instance set is generally as hard as finding a diverse double-colliding set of size m. If $\mathcal{T} = \{T_{i,j}\}_{i \in [k], j \in [m]}$ is a consistent (k, m)-deflate_slow instance set, then by definition, for each $i \in [k]$, $\{T_{1,j}\,[1 : \mathsf{min_match} + 1]\}_{j \in [m]}$ is a diverse compact-double-colliding set of

[95] Here a weaker property would be sufficient, but the stronger assumption can simplify our analysis to some extent.

size m. For the other direction, one can generally construct a consistent (k, m)-deflate_slow instance set by appending random bytes to each byte string in a diverse double-colliding set of size m and checking whether condition 3 holds. We formalise this observation in Theorem 11 and give a proof. We note that the algorithm and parameters in our proof are chosen to simplify the proof and are not intended for use in practice, and that our proof is strictly speaking non-constructive. Note also that our templates all have additional restrictions on the instance set for it to be suitable; they are not covered by this theorem. In practice, however, the method of appending random bytes works well.

Theorem 11. *Let $k, m, \ell, \mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$, $\Sigma \subseteq \{0, \ldots, 255\}$, such that $|\Sigma| \geq \mathsf{min_match}+2$ and $(|\Sigma|-1)^{\ell-\mathsf{min_match}-2} \geq (km)^3(\ell+1)$. Let H be a DEFLATE hash function with the parameters $\mathsf{min_match}$ and $\mathsf{hash_bits}$. If there exists a diverse double-colliding set with regard to H on Σ of size m, then there exists a consistent (k, m)-deflate_slow instance set with regard to H on Σ, in which every element is of size ℓ.*

Proof. Let $\{W_1, \ldots, W_m\} \subset \Sigma^*$ be a diverse double-colliding set with regard to H of size m. For each $i \in [m]$, let $W'_i = W_i[1 : \mathsf{min_match} + 1]$. By definition, the set $\{W'_1, \ldots, W'_m\}$ is diverse compact-double-colliding with regard to H.

Consider the set $\mathcal{T} = \{T_{i,j}\}_{i\in[k], j\in[m]} \subset \Sigma^*$ constructed using the following randomised algorithm. Let $\ell' = \ell - \mathsf{min_match} - 1$. Note that $(|\Sigma|-1)^{\ell-\mathsf{min_match}-2} \geq (km)^3(\ell+1)$ implies $\ell > \mathsf{min_match}+2$. For each $i \in [k]$ and $j \in [m]$, let $T_{i,j} = W'_j \| Z_{i,j}$, where $Z_{i,j}$ is sampled uniformly at random from the set

$$\mathcal{Z}_{i,j} = \left\{ Z \in \Sigma^{\ell'} \mid Z[|Z|] \notin \left(\{W'_j[\iota]\}_{\iota\in[\mathsf{min_match}+1]} \cup \{Z[\iota]\}_{\iota\in[|Z|-1]} \right) \right\};$$

that is, $Z_{i,j}$ is sampled uniformly at random from $\Sigma^{\ell'}$, conditioned on that the last byte of $Z_{i,j}$ does not appear elsewhere in T_j. Observe that $|\mathcal{Z}_{i,j}| > 0$ for each $(i, j) \in [k] \times [m]$ since $|\Sigma| > \mathsf{min_match} + 2$. Conditions 1 and 2 in Definition 10 trivially hold for $\mathcal{T}$. In the rest of the proof, we show that the probability that $|\mathcal{T}| < km$ or condition 3 in Definition 10 does not hold for $\mathcal{T}$ is strictly smaller than 1, thereby proving the existence of a consistent (k, m)-deflate_slow instance set.

For each $(i_1, j_1), (i_3, j_3), (i_4, j_4) \in [k]\times[m]$, let $T_\iota = T_{(i_\iota, j_\iota)}$ for $\iota \in \{1, 3, 4\}$, and let $E = E_{((i_\iota, j_\iota))_{\iota\in\{1,3,4\}}}$ denote the event that there exists some $(i_2, j_2) \in [k] \times [m]$ such that $(i_1, j_1) \neq (i_3, j_3)$ or $j_2 \neq j_4$, and $T_1[2 : |T_1|] \| T_{i_2,j_2}[1]$ is a substring of $T_3 \| T_4$. We consider the four possible cases as follows.

1. $(i_1, j_1) \neq (i_3, j_3)$ and $(i_1, j_1) \neq (i_4, j_4)$. We have

$$\Pr[E] \leq \Pr[Z_{i_1,j_1} \text{ is a substring of } (T_3\|T_4)[\mathsf{min_match} + 1 : |T_3| + |T_4| - 1]] \leq \frac{\ell+1}{|\mathcal{Z}_{i_1,j_1}|}.$$

2. $(i_1, j_1) \neq (i_3, j_3)$ and $(i_1, j_1) = (i_4, j_4)$. We have $T_1 = T_4$. Suppose $T_1[2 : |T_1|] \| T_{i_2,j_2}[1]$ is a substring of $T_3 \| T_1$ for some $(i_2, j_2) \in [k] \times [m]$, then it must be that $T_1[2 : |T_1|]$ is a substring of T_3, because $T_1[|T_1|]$ does not appear elsewhere in T_1. We have

$$\Pr[E] \leq \Pr[Z_{i_1,j_1} \text{ is a substring of } T_3[\mathsf{min_match} + 1 : |T_3|]] \leq \frac{2}{|\mathcal{Z}_{i_1,j_1}|}.$$

3. $(i_1, j_1) = (i_3, j_3)$ and $(i_1, j_1) \neq (i_4, j_4)$. We have $T_1 = T_3$. Suppose $T_1[2 : |T_1|] \| T_{i_2,j_2}[1]$ is a substring of $T_1 \| T_4$ for some $(i_2, j_2) \in [k] \times [m]$ where $j_2 \neq j_4$, then it must be that $T_1[2 : |T_1|]$ is a substring of T_4, because $T_1[|T_1|]$ does not appear elsewhere in T_1, and $T_{i_2,j_2}[1] \neq T_4[1]$. We have

$$\Pr[E] \leq \Pr[Z_{i_1,j_1} \text{ is a substring of } T_4[\mathsf{min_match} + 1 : |T_4|]] \leq \frac{2}{|\mathcal{Z}_{i_1,j_1}|}.$$

4. $(i_1, j_1) = (i_3, j_3)$ and $(i_1, j_1) = (i_4, j_4)$. We have $T_1 = T_3 = T_4$. Suppose, for a contradiction, that $T_1[2 : |T_1|] \| T_{i_2,j_2}[1]$ is a substring of $T_1 \| T_1$ for some $(i_2, j_2) \in [k] \times [m]$ where $j_2 \neq j_4$, then it must be that $T_1[2 : |T_1|] \| T_{i_2,j_2}[1] = (T_1\|T_1)[2 : |T| + 1]$, because $T_1[|T_1|]$ does not appear elsewhere in T_1. We have $T_{i_2,j_2}[1] = T_1[1]$ and consequently $j_2 = j_1$ by condition 1, contradictory to our assumptions that $j_1 = j_4$ and $j_2 \neq j_4$. Therefore, $\Pr[E] = 0$.

Note that, if $|\mathcal{T}| < km$, then there exist some indices $(i_1^*, j_1^*) \neq (i_3^*, j_3^*)$ such that $T_1 = T_3$, so for every index (i_4^*, j_4^*), we have $\Pr\left[E_{((i_1^*,j_1^*),(i_3^*,j_3^*),(i_4^*,j_4^*))}\right] = 1$.

Applying a union bound over all possible cases, we get

$$\begin{aligned}
&\Pr[|\mathcal{T}| < km \text{ or condition 3 does not hold for } \mathcal{T}] \\
&\leq \sum_{(i_1,j_1)\in[k]\times[m]} \sum_{\substack{(i_3,j_3)\in[k]\times[m] \\ (i_1,j_1)\neq(i_3,j_3)}} \sum_{(i_4,j_4)\in[k]\times[m]} \Pr\left[E_{((i_1,j_1),(i_3,j_3),(i_4,j_4))}\right] \\
&< \sum_{(i_1,j_1)\in[k]\times[m]} (km)^2 \frac{\ell+1}{|\mathcal{Z}_{i_1,j_1}|} \\
&\leq \frac{(km)^3(\ell+1)}{(|\Sigma|-1)^{\ell-\mathsf{min_match}-2}}.
\end{aligned}$$

Therefore,

$$\Pr[\mathcal{T} \text{ is a consistent } (k,m)\text{-}\mathsf{deflate_slow} \text{ instance set}] \;>\; 1 - \frac{(km)^3(\ell+1)}{(|\Sigma|-1)^{\ell-\mathsf{min_match}-2}} \;\geq\; 0. \qquad \square$$

We remark that one can potentially improve the bound in Theorem 11 using the Lovász Local Lemma.

7.3.2 Main template

We mainly focus on the case where the attacker can find a diverse double-colliding set on Σ of $m \geq 3$ byte strings, which we expect to be true for most real-world DEFLATE implementations and configurations.

We describe a template for this setting. Our template requires a consistent (k,m)-$\mathsf{deflate_slow}$ instance set $\mathcal{T}$ on Σ with $m \geq 3$; as discussed above, one can generally construct such an instance set by appending random bytes to every byte string in a diverse double-colliding set on Σ of size m. We will specify more constraints on k, m and $\mathcal{T}$ later. Let v denote the hash value at which $\mathcal{T}$ is double-colliding.

Construction. The query Q is constructed from the (k,m)-$\mathsf{deflate_slow}$ instance set $\mathcal{T}$ as follows. For $Q = \mathsf{dict}_1 \| \mathsf{filler} \| \mathsf{dict}_2 \| P \| g \| S$, dict_1 is the empty string ε and dict_2 represents the sequence of two byte strings $(S[0] \| S_1 \| S_2[1 : |S_2| - 1], T_{k,m})$ (or vice versa),[96] the filler is the empty string ε, and the chain is $S = (S_1 \| S_2)^n$, where $n \in \mathbb{N}_+$,

$$\begin{aligned}
S_1 &= T_{1,2} \| T_{1,1} \| \cdots \| T_{1,m} \| T_{1,(m-1)} \| \cdots \| T_{k,2} \| T_{k,1} \| \cdots \| T_{k,m} \| T_{k,(m-1)} \\
S_2 &= T_{1,1} \| T_{1,2} \| \cdots \| T_{1,(m-1)} \| T_{1,m} \| \cdots \| T_{k,1} \| T_{k,2} \| \cdots \| T_{k,(m-1)} \| T_{k,m}
\end{aligned}$$

if m is even, and

$$\begin{aligned}
S_1 &= T_{1,1} \| T_{1,3} \| T_{1,2} \| \cdots \| T_{1,m} \| T_{1,(m-1)} \| \cdots \| T_{k,1} \| T_{k,3} \| T_{k,2} \| \cdots \| T_{k,m} \| T_{k,(m-1)} \\
S_2 &= T_{1,1} \| T_{1,2} \| T_{1,3} \| \cdots \| T_{1,(m-1)} \| T_{1,m} \| \cdots \| T_{k,1} \| T_{k,2} \| T_{k,3} \| \cdots \| T_{k,(m-1)} \| T_{k,m}
\end{aligned}$$

if m is odd. Note that S_2 is a permutation of S_1 with respect to the byte strings in $\mathcal{T}$ in both cases, and therefore $|S_1| = |S_2|$.

Distinguished substring. To simplify our description of the requirements and analysis, we first introduce the concept of distinguished substrings. Recall from Definition 4 that a collision counter $cc_{\mathsf{H}}(W,v)$ counts the number of collisions at v with regard to H within a byte string W.

Definition 12 (Distinguished hash value). *Let* $\mathsf{min_match}, \mathsf{hash_bits}, \mathsf{max_chain} \in \mathbb{N}_+$*, where* $\mathsf{min_match} \geq 2$*. Let* H *be a DEFLATE hash function with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$*. Let* Q *be an instantiation of a template with* $S = (S_1 \| S_2)^n$ *using an instance set* $\mathcal{T}$ *with respect to* H *and the parameters* $\mathsf{min_match}$ *and* $\mathsf{max_chain}$*, where* $S[0] = S_2[|S_2|]$*.*[97] *An integer* $v' \in \{0, \ldots, 2^{\mathsf{hash_bits}} - 1\}$ *is a distinguished hash value with regard to the query* Q*, the hash function* H *and the parameters* $\mathsf{min_match}$ *and* $\mathsf{max_chain}$*, or distinguished in short, if:*

1. $cc_{\mathsf{H}}(Q,v') \leq cc_{\mathsf{H}}(S[0] \| (S_1 \| S_2)^{n-1}, v') + \mathsf{max_chain}$*;*

[96] Note that, if $\mathsf{max_match}$ divides $|S_1| + |S_2|$, then the dictionary can instead represent a single byte string $S[0] \| S_1 \| S_2$; for simplicity, we do not discuss this variant in our requirements and analysis.

[97] All of our templates require that $S[0] = S_2[|S_2|]$; we will discuss this in more detail in Section 7.3.2.

2. $cc_{\mathsf{H}}(S_1\|S_2\|S_1\|S_2, v') \leq \mathsf{max_chain}$.

Definition 13 (Distinguished substring). *Let* $\mathsf{min_match}, \mathsf{hash_bits}, \mathsf{max_chain}, \mathsf{nice_length} \in \mathbb{N}_+$, *where* $\mathsf{nice_length} \geq \mathsf{min_match} \geq 2$. *Let* H *be a DEFLATE hash function with the parameters* $\mathsf{min_match}$ *and* $\mathsf{hash_bits}$. *Let* Q *be an instantiation of a template with* $S = (S_1\|S_2)^n$ *using an instance set* $\mathcal{T}$ *with respect to* H *and the parameters* $\mathsf{min_match}$ *and* $\mathsf{max_chain}$, *where* $S[0] = S_2[|S_2|]$. *A byte string* Z *with hash value* $v' := \mathsf{H}(Z[1 : \mathsf{min_match}])$ *is a distinguished substring with regard to the query* Q, *the hash function* H *and the parameters* $\mathsf{min_match}$, $\mathsf{max_chain}$ *and* $\mathsf{nice_length}$, *or distinguished in short, if:*

1. Z *is a substring of* $S[0]\|S_1\|S_2$;
2. Z *appears in* Q *as a substring for exactly* $n+1$ *times, i.e.* $\{i \in [|Q|-|Z|+1] \mid Z = Q[i : i+|Z|-1]\} = n+1$;
3. v' *is a distinguished hash value with regard to* Q, H, $\mathsf{min_match}$ *and* $\mathsf{max_chain}$;
4. $|Z| \leq \mathsf{nice_length}$.[98]

By conditions 1 and 2 and the construction of the query Q, we know that a distinguished substring Z appears once outside of $S[0]\|S$ (in one of the dictionaries except for the corner case where Z is a long suffix of $S[0]\|S_1\|S_2$), and that Z appears n times in $S[0]\|S$, whose pairwise distances are multiples of $|S_1| + |S_2|$. These conditions ensure that no unwanted matches for Z are present in the query Q. By restricting the number of collisions at v' and the length of Z, conditions 3 and 4 ensure that, if the LZ77 pointer is at the start of some occurrence of Z in the query, then LZ77 can find a match starting at the previous occurrence of Z in the query, if any.[99] More specifically, condition 1 in Definition 12 guarantees that there are fewer than $\mathsf{max_chain}$ collisions at v' between Z in one of the dictionaries and the first Z in $S[0]\|S$, condition 2 in Definition 12 guarantees that there are fewer than $\mathsf{max_chain}$ collisions between every neighbouring occurrences of Z in $S[0]\|S$, and condition 4 here guarantees that LZ77 does not stop looking for a match prematurely after having found a strict prefix of Z on the hash chain.

For simplicity, Definitions 12 and 13 both assume n to be fixed. However, it is straightforward to see that a hash value is distinguished for all $n \geq n_0$ if it is distinguished when $n = n_0$ for some $n_0 \geq 2$, and that a substring is distinguished for all $n \geq n_1$ if it is distinguished when $n = n_1$ for some $n_1 \geq 3$.

In practice, requirements on hash values or substrings being distinguished usually do not need to be enforced. The main reason is that our constructions generally tolerate minor violations to the above definitions, in which we assume ideal patterns to simply our requirements and analysis.

Requirements. For this template to work correctly, we require that $S[0] = T_{k,m}[|T_{k,m}|]$ and the byte string $S[0 : \mathsf{min_match} - 1]$ be distinguished. We will see in Section 8.2.2 that one can change $S[0]$ to an arbitrary byte,[100] so there is no need to modify the instance set $\mathcal{T}$ based on the guess g in practice.

For constraints on k and m, we first discuss the special case where the only collisions at v in $(S_1\|S_2)^*$ come from $T_{i,j}[1 : \mathsf{min_match}]$ and $T_{i,j}[2 : \mathsf{min_match} + 1]$ for every $i \in [k]$ and $j \in [m]$. This is likely already true for a randomly chosen instance set $\mathcal{T}$ if km is not too large, and even more so if one carefully chooses $\mathcal{T}$ or rejects nonconforming candidates for $\mathcal{T}$. We require that

$$\frac{\mathsf{max_chain}}{4} < km \leq \frac{\mathsf{max_chain}}{2} - 2.$$

For the general case where additional collisions may be present, let $S' = S_1\|S_2\|S_1\|S_2$,

$$c_{1,v} = \max_{i \in [|S_1|+|S_2|]} cc_{\mathsf{H}}\left(S'\left[i : i + |S_1| + \max_{T \in \mathcal{T}}|T| + \mathsf{min_match} - 2\right], v\right),$$

which is an upper bound on the maximum number of collisions at v between two consecutive occurrences of a string $T \in \mathcal{T}$, and

$$c_{2,v} = 2\sum_{i=1}^{k}\sum_{j=1}^{m} cc_{\mathsf{H}}(T_{i,j}, v),$$

[98] For simplicity, we typically only consider the case where $|Z| = \mathsf{min_match}$.

[99] Formally, we also need to consider the sliding window size $\mathsf{window_size}$; we will make a separate assumption in our requirements to address this.

[100] This can be achieved by prepending a NOP gadget to the chain S. Even in the worst case where all possible choices of $S[0]$ and $T_{1,2-(m \bmod 2)}$ result in collisions, we can still slightly modify the template to make it work; one idea is to first enlarge the difference in the LZ77 pointer positions depending on whether $ck[1] = g$ to more than one byte, and then perform collision-based amplification. We do not discuss this case as it is almost impossible for this case to occur in practice.

which is a lower bound on the minimum number of collisions at v from a string $T \in \mathcal{T}$ in S_1 (resp. S_2) to its previous occurrence in S_1 (resp. S_2). By definition, $cc_{\mathsf{H}}(T_{i,j}, v) \geq 2$ for each $i \in [k]$ and $j \in [m]$, $c_{1,v} \geq 2km + 2$, and $c_{2,v} \geq 4km$. We require that[101]

$$c_{1,v} + cc_{\mathsf{H}}(T_{k,m}, v) \leq \mathsf{max_chain} < c_{2,v}.$$

In order for our construction to work properly even when n is extremely large, we require that at least one of the two following constraints on the instance set $\mathcal{T}$ hold:

(i) There exists $\tau \in \mathbb{N}_+$ and $k_0, \ldots, k_\tau \in \mathbb{N}$, where $\tau < \frac{\mathsf{window_size}}{2 \cdot \mathsf{max_match}}$ and $0 = k_0 < k_1 < \cdots < k_\tau = k$, such that for every $t \in [\tau]$,

$$\sum_{i=k_{t-1}+1}^{k_t} \sum_{j=1}^{m} |T_{i,j}| = \mathsf{max_match},$$

and if $t < \tau$,

$$T_{k_t,(m-1)} \left[\left|T_{k_t,(m-1)}\right|\right] \| T_{((k_t+1),2-(m \bmod 2))} [1 : \mathsf{min_match} - 1]$$

and

$$T_{k_t,m} [|T_{k_t,m}|] \| T_{(k_t+1),1} [1 : \mathsf{min_match} - 1]$$

are distinguished. In addition, we require that $T_{k,(m-1)}[\left|T_{k,(m-1)}\right|] \| T_{1,1} [1 : \mathsf{min_match} - 1]$ be distinguished.

(ii) The integer $|S_1| + |S_2|$ divides $2 \cdot \mathsf{max_match}$ (or, equivalently, $|S_1| = |S_2|$ divides $\mathsf{max_match}$),[102] $\mathsf{max_match}$ divides $|S| - |S_1| - |S_2|$, and, if $|S_1| + |S_2|$ does not divide $\mathsf{max_match}$, then we require that $T_{k,(m-1)}[\left|T_{k,(m-1)}\right|] \| T_{1,1} [1 : \mathsf{min_match} - 1])$ be distinguished.

Note that constraint (i) implies $|S_1| + |S_2| = 2\tau \cdot \mathsf{max_match}$.

Finally, for our analysis, we also make the following assumptions, but they normally do not need to be enforced in practice or can be significantly relaxed.

1. The only substrings with the hash value v in the query are entirely contained within the dictionary entries and the chain S; more formally,

$$cc_{\mathsf{H}}(Q, v) = cc_{\mathsf{H}}\left(S[0] \| S_1 \| S_2 [1 : |S_2| - 1], v\right) + cc_{\mathsf{H}}(T_{k,m}, v) + cc_{\mathsf{H}}(S, v).$$

   A necessary but not sufficient condition is that $cc_{\mathsf{H}}(P \| g) = 0$.
2. If $|S_1| + |S_2| < \mathsf{nice_length}$, then the string $S[0] \| S_1 \| S_2$ does not appear in the sliding window before the query Q.
3. For every $T_3, T_4 \in \mathcal{T}$, where $T_{k,m} \neq T_3$ or $S[\ell_S + 1] \neq T_4[1]$, it holds that $T_{k,m} [2 : |T_{k,m}|] \| S[\ell_S + 1]$ is not a substring of $T_3 \| T_4$. This assumption ensures that LZ77 does not find a match longer than $T_{k,m}$ at the end of the query when the guess is correct. This assumption can be relaxed, and it has almost no effect on the compressed length difference.
4. It holds that $|\mathsf{dict}_1 \| \mathsf{filler} \| \mathsf{dict}_2 \| P \| g \| S_1 \| S_2| < \mathsf{window_size}$, which implies $|S_1| + |S_2| < \mathsf{window_size}/2$. This assumption is generally satisfied trivially unless $\mathsf{window_size}$ is small or $\mathsf{max_chain}$ is large, and it can be further relaxed.

Analysis. We are now ready to analyse the behaviour and output of LZ77 for $P \| g \| S$ in the query. To simplify our arguments, we only analyse the case where m is even; the case where m is odd follows almost identically. We assume that LZ77 finds a match for $P \| g$ in the sliding window and moves the pointer to $S[1]$ in the query Q if $ck[1] = g$, and LZ77 finds a match for P in the sliding window and moves the pointer to $S[0] = g$ in the query Q otherwise.

– *The guess is correct;* i.e. $ck[1] = g$. Whether constraint (i) or (ii) above holds for the instance set $\mathcal{T}$ is not important for this case. By assumption, the LZ77 pointer is at $S[1] = S_1[1]$, the first byte of $T_{1,2}$. LZ77 looks for a match at this position by checking the entries of the hash chain corresponding to the hash value of $T_{1,2} [1 : \mathsf{min_match}]$, which is v by definition. We claim that LZ77 finds a match

[101] Here one should interpret the bounds as constraints on k, m and $\mathcal{T}$, rather than just on k and m.

[102] We can modify the template to drop this requirement if it is difficult to meet (e.g. $\mathsf{max_match}$ is prime); for example, we can use S_1 and S_2 of different lengths by repeating some $T \in \mathcal{T}$, as shown in one of our methods for advanced bootstrapping in Section 7.2.5. However, all DEFLATE compressors we surveyed have fixed the default value of $\mathsf{max_match}$ to 258, the largest value allowed in the DEFLATE specification.

for $T_{1,2}$ and moves the pointer to $S[|T_{1,2}|+1]$, the first byte of $T_{1,1}$ in S_1. To prove this claim, we argue as follows.

By our assumptions, LZ77 finds a match for $T_{1,2}$ at $S_2[|T_{1,1}|+1]$ in one of the dictionaries, or possibly at a position closer to S, after checking at most $c_{1,v} + cc_{\mathsf{H}}(T_{k,m}, v) \leq \mathsf{max_chain}$ entries of the hash chain. It remains to show that LZ77 cannot extend the match for $T_{1,2}$ or improve it by lazy matching. For this purpose, we show that the substring $T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1]$ only appears as $S_1[2:|T_{1,2}|+1]$ in every S_1 and not anywhere else in the query. If this is true, then LZ77 can only find a match for $T_{1,2}\|T_{1,1}[1]$ or $T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1:2]$ at the start of S_1 in one of the dictionaries, which is unreachable when the pointer is at $S[1]$ or $S[2]$ as it would require checking at least $c_{2,v} > \mathsf{max_chain}$ entries, and therefore the match cannot be extended or improved by lazy matching.

Suppose $T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1]$ appears in $(S_1\|S_2)^*$ as a substring of $T'_1\|\cdots\|T'_\nu$, the concatenation of a sequence of strings $(T'_1,\ldots,T'_\nu) \in \mathcal{T}^\nu$ for some $\nu \in \mathbb{N}_+$, and the sequence is minimal in the sense that $T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1]$ is not a substring of $T'_1\|\cdots\|T'_{\nu-1}$ or $T'_2\|\cdots\|T'_\nu$. We discuss three different cases for ν. Recall from Definition 10 that, for every $T_1,T_2,T_3,T_4 \in \mathcal{T}$, where $T_1 \neq T_3$ or $T_2[1] \neq T_4[1]$, it holds that $T_1[2:|T_1|]\|T_2[1]$ is not a substring of $T_3\|T_4$.

- If $\nu = 1$, then $T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1]$ is a substring of T'_1. We let $T_1 = T_{1,2}$, $T_2 = T_{1,1}$, $T_3 = T'_1$, and $T_4 = T_{1,2}$. By definition, $T_2[1] \neq T_4[1]$ and $T_1[2:|T_1|]\|T_2[1]$ is a substring of $T_3\|T_4$, contradictory to the fact that $\mathcal{T}$ is a $\mathsf{deflate_slow}$ instance set.
- If $\nu = 2$, then $T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1]$ is a substring of $T'_1\|T'_2$. Applying Definition 10 to $T_1 = T_{1,2}$, $T_2 = T_{1,1}$, $T_3 = T'_1$, and $T_4 = T'_2$ yields $T_{1,2} = T'_1$ and $T_{1,1}[1] = T'_2[1]$.
- If $\nu > 2$, then $T'_1[|T'_1|]\|T'_2\|\cdots\|T'_{\nu-1}\|T'_\nu[1]$ is a substring of $T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1]$. The intuition is that $T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1]$ starts in T'_1 and ends in T'_ν. Let $W = T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1]$ and $Z = T'_1\|\cdots\|T'_\nu$; since W is a substring of Z, there exists some $d \in [|Z|-|W|+1]$ such that $W = Z[d:d+|W|-1]$. By our assumption on the minimality of $(T'_1,\ldots,T'_\nu)$, we have $d \leq |T_1|$ and $d+|W|-1 \geq |Z|-|T'_\nu|+1$, so $T'_1[|T'_1|]\|T'_2\|\cdots\|T'_{\nu-1}\|T'_\nu[1]$ is a substring of W.

  An immediate corollary is that $T'_2\|T'_3[1]$ is a substring of $T_{1,2}\|T_{1,1}$. Another corollary is that $|T'_2| < |T_{1,2}|$, and it follows that $T'_2 \neq T_{1,2}$. Let $T_1 = T'_2$, $T_2 = T'_3$, $T_3 = T_{1,2}$, and $T_4 = T_{1,1}$. We have $T_1 \neq T_3$ and $T_1[2:|T_1|]\|T_2[1]$ is a substring of $T_3\|T_4$, contradictory to the fact that $\mathcal{T}$ is a $\mathsf{deflate_slow}$ instance set.

To summarise, it must be that $\nu = 2$, $T'_1 = T_{1,2}$ and $T'_2[1] = T_{1,1}[1]$. Because $T_{1,2}$ is followed by $T_{1,3}$ in S_2, where $T_{1,1}[1] \neq T_{1,3}[1]$ by definition, such a case only occurs in S_1. Therefore, the substring $T_{1,2}[2:|T_{1,2}|]\|T_{1,1}[1]$ only appears as $S_1[2:|T_{1,2}|+1]$ in $(S_1\|S_2)^*$, and similarly in the query. We conclude that LZ77 finds a match for $T_{1,2}$ and moves the pointer to the first byte of $T_{1,1}$.

By repeating similar arguments, we can show that LZ77 subsequently finds matches for $T_{1,1},\ldots,T_{1,m}$, $T_{1,(m-1)},\ldots,T_{k,2},T_{k,1},\ldots,T_{k,m},T_{k,(m-1)}$ in the dictionaries and moves the pointer to $S[|S_1|+1] = S_2[1]$, the first byte of $T_{1,1}$ in S_2, and then LZ77 finds matches for $T_{1,1},T_{1,2},\ldots,T_{1,(m-1)},T_{1,m},\ldots,T_{k,1},T_{k,2},\ldots,T_{k,(m-1)},T_{k,m}$, likely in the first occurrence of S_1 in S, and moves the pointer to $S[|S_1|+|S_2|+1]$, the first byte of $T_{1,2}$ in the second occurrence of S_1 in S. Continuing this process, we conclude that LZ77 encodes $P\|g\|S$ as $2kmn+1$ back-references, outputs

$$\texttt{<L}(\ell_P+1)\texttt{,D*>}\|\left(\texttt{<L}|T_{1,2}|\texttt{,D*>}\|\cdots\|\texttt{<L}\left|T_{k,(m-1)}\right|\texttt{,D*>}\|\texttt{<L}|T_{1,1}|\texttt{,D*>}\|\cdots\|\texttt{<L}|T_{k,m}|\texttt{,D*>}\right)^n$$

and moves the pointer to $S[\ell_S+1]$, the byte right after the chain S.[103]

– *The guess is incorrect;* i.e. $ck[1] \neq g$. By assumption, the LZ77 pointer is at $S[0] = g$, just before the chain S. We discuss two possible cases depending on which constraint on the instance set $\mathcal{T}$ holds:

- *Constraint (i) holds.* Recall that there exists $\tau \in \mathbb{N}_+$ and $k_0,\ldots,k_\tau \in \mathbb{N}$, where $\tau < \frac{\mathsf{window_size}}{2\cdot\mathsf{max_match}}$ and $0 = k_0 < k_1 < \cdots < k_\tau = k$, such that for every $t \in [\tau]$,

$$\sum_{i=k_{t-1}+1}^{k_t}\sum_{j=1}^{m}|T_{i,j}| = \mathsf{max_match}.$$

  LZ77 looks for a match at $S[0]$ by checking the hash chain for $S[0:\mathsf{min_match}-1]$, which is distinguished by assumption. LZ77 finds a match $S[0]\|S[1:\mathsf{max_match}-1]$ in one of the dictionaries, which is equivalent to

$$S[0]\|T_{1,2}\|T_{1,1}\|\cdots\|T_{1,m}\|T_{1,(m-1)}\|\cdots\|T_{k_1,2}\|T_{k_1,1}\|\cdots\|T_{k_1,m}\|T_{k_1,(m-1)}\left[1:\left|T_{k_1,(m-1)}\right|-1\right].$$

[103] Note that the analysis for the last back-reference for $T_{k,m}$ at the end of the chain S relies on assumption 3.

The match is of $\mathsf{max_match}$ bytes, which means it cannot be extended or improved by lazy matching. Therefore, LZ77 outputs a back-reference for $S[0]\|S\left[1:\mathsf{max_match}-1\right]$ and moves the pointer to $S[\mathsf{max_match}] = S_1[\mathsf{max_match}] = T_{k_1,(m-1)}[\left|T_{k_1,(m-1)}\right|]$.
LZ77 then looks for a match at the current pointer position by checking the hash chain for $S\left[\mathsf{max_match}:\mathsf{max_match}+\mathsf{min_match}-1\right]$, which is

$$T_{k_1,(m-1)}[\left|T_{k_1,(m-1)}\right|]\|T_{(k_1+1),2}\left[1:\mathsf{min_match}-1\right]$$

if $\tau > 1$ and

$$T_{k,(m-1)}[\left|T_{k,(m-1)}\right|]\|T_{1,1}\left[1:\mathsf{min_match}-1\right]$$

otherwise, both of which are distinguished by assumption. By similar arguments as above, LZ77 outputs a back-reference for $S\left[\mathsf{max_match}:2\cdot\mathsf{max_match}-1\right]$ to one of the dictionaries and moves the pointer to $S[2\cdot\mathsf{max_match}]$.
Continuing this process, we conclude that LZ77 outputs $2\tau n+1 = \ell_S/\mathsf{max_match}+1$ back-references

$$\texttt{<L}\ell_P\texttt{,D*>}\|\left(\texttt{<L}\mathsf{max_match}\texttt{,D*>}\right)^{2\tau})^n,$$

where the $\mathsf{max_match}$-byte back-references either point to one of the dictionaries or previous positions in S, and moves the pointer to $S[\ell_S]$, the last byte of the chain S.

- *Constraint (ii) holds.* Recall that $|S_1|+|S_2|$ divides $2\cdot\mathsf{max_match}$ and $\mathsf{max_match}$ divides $|S|-|S_1|-|S_2|$. If $|S_1|+|S_2| = 2\cdot\mathsf{max_match}$, then constraint (i) also holds (with $\tau=1$); suppose hereafter that $|S_1|+|S_2| < 2\cdot\mathsf{max_match}$. Let $\omega = 2\cdot\mathsf{max_match}/(|S_1|+|S_2|)$; by definition, $\omega\in\mathbb{N}_+\backslash\{1\}$, and it follows that $|S_1|+|S_2|\leq\mathsf{max_match}$. Let $\kappa = (|S|-|S_1|-|S_2|)/\mathsf{max_match}$. We have $\kappa\in\mathbb{N}$ and $\kappa = (n-1)(|S_1|+|S_2|)/\mathsf{max_match} = 2(n-1)/\omega$.
LZ77 looks for a match at $S[0]$ by checking the hash chain for $S\left[0:\mathsf{min_match}-1\right]$, which is distinguished by assumption. LZ77 finds a match $S[0]\|S_1\|S_2\left[1:|S_2|-1\right]$ of $|S_1|+|S_2|\leq\mathsf{max_match}$ bytes. LZ77 cannot find a longer match in the sliding window by assumption. To see that the match cannot be improved by lazy matching, note that we have shown in our analysis for the case where $ck[1]=g$ that LZ77 cannot find a match longer than $|T_{1,2}|<|S_1|$ at $S[1]$. Therefore, LZ77 outputs a back-reference for $S[0]\|S_1\|S_2\left[1:|S_2|-1\right]$ and moves the pointer to $S[|S_1|+|S_2|] = S_2[|S_2|] = T_{k,m}[|T_{k,m}|]$.
If $\kappa=0$, then the LZ77 pointer has reached the last byte of the query; assume hereafter that $\kappa\geq 1$. There are $\kappa\cdot\mathsf{max_match}$ bytes in the query after the current pointer position. LZ77 looks for a match at the current pointer position by checking the hash chain for

$$S_2[|S_2|]\|S_1\left[1:\mathsf{min_match}-1\right] = S[0]\|T_{1,2}\left[1:\mathsf{min_match}-1\right],$$

which is distinguished by assumption; recall that $S_2[|S_2|] = T_{k,m}[|T_{k,m}|] = S[0]$. Because $S\left[|S_1|+|S_2|:|S|\right] = S\left[0:|S|-|S_1|-|S_2|\right]$ and LZ77 can find a back-reference that overlaps with the matched substring, by our assumptions, LZ77 finds a match $S\left[0:\mathsf{max_match}-1\right]$, which is

$$S[0]\|(S_1\|S_2)^{\frac{\omega}{2}-1}\|S_1\|S_2\left[1:|S_2|-1\right]$$

if ω is even, and

$$S[0]\|(S_1\|S_2)^{\frac{\omega-1}{2}}\|S_1\left[1:|S_1|-1\right]$$

if ω is odd; recall that $|S_1|=|S_2|$. The match is of $\mathsf{max_match}$ bytes, which means it cannot be extended or improved by lazy matching. Therefore, LZ77 outputs a back-reference for $S\left[0:\mathsf{max_match}-1\right]$ and moves the pointer to $S[|S_1|+|S_2|+\mathsf{max_match}]$, which is $S_2[|S_2|]$ if ω is even and $S_1[|S_1|]$ otherwise.
It is straightforward to see that we can repeat similar arguments κ times in total to show that LZ77 outputs a series of $\mathsf{max_match}$-byte back-references if ω is even. So we now focus on the case where ω is odd. If ω is odd, then $|S_1|+|S_2|$ does not divide $\mathsf{max_match}$ by definition, and $\kappa = 2(n-1)/\omega$ is even. LZ77 looks for a match at $S[|S_1|+|S_2|+\mathsf{max_match}]$ by checking the hash chain for

$$S_1[|S_1|]\|S_2\left[1:\mathsf{min_match}-1\right] = T_{k,(m-1)}\left[\left|T_{k,(m-1)}\right|\right]\|T_{1,1}\left[1:\mathsf{min_match}-1\right],$$

which is distinguished by assumption. Similar to the arguments above, LZ77 outputs a back-reference of $\mathsf{max_match}$ bytes for

$$S_1[|S_1|]\|(S_2\|S_1)^{\frac{\omega-1}{2}}\|S_2\left[1:|S_2|-1\right]$$

and moves the pointer to $S[|S_1| + |S_2| + 2 \cdot \mathsf{max_match}] = S_2[|S_2|]$. Continuing this process, we see that LZ77 also outputs a series of $\mathsf{max_match}$-byte back-references when ω is odd.
We conclude that the LZ77 algorithm outputs $\kappa + 2 = \lceil \ell_S / \mathsf{max_match} \rceil + 1$ back-references

$$\texttt{<L}\ell_P\texttt{,D*>}\|\texttt{<L}(|S_1| + |S_2|)\texttt{,D*>}\|(\texttt{<L}\mathsf{max_match}\texttt{,D*>})^{\kappa},$$

where the $\mathsf{max_match}$-byte back-references point to the previous positions in S, and moves the pointer to $S[\ell_S]$, the last byte of the chain S.

Performance. If $ck[1] = g$, then the LZ77 output for $P\|g\|S$ consists of $kmn + 1$ back-references; otherwise, the LZ77 output only consists of $\lceil \ell_S / \mathsf{max_match} \rceil + 1$ back-references, and likely one literal for the last byte $S[\ell_S]$. Let $\ell = (\sum_{T \in \mathcal{T}} |T|)/(km)$ be the average length of the byte strings in the instance set $\mathcal{T}$; by definition, $\ell \geq \mathsf{min_match}+1$, and the equality holds if $\mathcal{T}$ is compact-double-colliding. The length of the chain is $\ell_S = 2kmn\ell \geq 2kmn(\mathsf{min_match}+1)$, and the length of the query is $\ell_Q = 2km\ell(n+1)+\ell_P+c$, where c is a small constant. The ratio between the LZ77 output length difference and the query length is around $1/\ell - 1/\mathsf{max_match}$, where ℓ only depends on the instance set $\mathcal{T}$ by definition. Therefore, the LZ77 output length difference grows about linearly to the query length.

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 expected to be proportional to

$$2kmn - \left\lceil \frac{\ell_S}{\mathsf{max_match}} \right\rceil \approx 2kmn \left(1 - \frac{\ell}{\mathsf{max_match}}\right).$$

Because n can be arbitrarily large, the compressed length difference can also be made arbitrarily large.

Example. We demonstrate how to construct a collision-based amplification query from a double-colliding set using the main template. Our implementation can automate this process; see Section 9.1. As with Section 7.2, we use our example setting, where we set the zlib compression level to 4 and other parameters to default. We assume that the guess for $ck[1]$ is $g = \texttt{1}$.

Our starting point is $\{\texttt{0pqQ}, \texttt{1PqQ}, \texttt{20qQ}\}$, a diverse double-colliding set of $m = 3$ byte strings at hash value 20081. We first build a suitable (k, m)-$\mathsf{deflate_slow}$ instance set $\mathcal{T}$ by extending the byte strings in the double-colliding set. When building the instance set $\mathcal{T}$, we need to take into account the requirements on $\mathcal{T}$, and, for better performance, we also want its average length $\ell = (\sum_{T \in \mathcal{T}} |T|)/(km)$ to be small. Specifically, we consider the following aspects:

- *Value of k.* We apply the bounds on km for the special case where no unintended collisions would be present in $(S_1\|S_2)^*$. With $\mathsf{max_chain} = 16$, we obtain $4 < km \leq 6$, so the only option is $k = 2$.
- *Total length.* Our instance set needs to meet at least one of the two constraints specified above. Because k and m are small, we consider constraint (ii), which requires that the total length of the instance set be a divisor of $\mathsf{max_match} = 258 = 2 \cdot 3 \cdot 43$. For better performance, we choose 43, the smallest divisor of $\mathsf{max_match}$ above $km(\mathsf{min_match} + 1) = 24$.
- *Individual lengths.* We need to decide on how to distribute the 19 extra bytes within the instance set. Since we do not need to meet constraint (i), we can simply distribute them as evenly as possible.
- *Tail byte.* We also require that $S[0] = T_{k,m}[|T_{k,m}|]$, which we meet here by setting $T_{2,3}[|T_{2,3}|] = \texttt{1}$.

We can then build $\mathcal{T}$ by appending the specified number of bytes to each byte string in the double-colliding set. For example, one can verify that $\{\texttt{0pqQAAA}, \texttt{1PqQAAA}, \texttt{20qQAAA}, \texttt{0pqQBBB}, \texttt{1PqQBBB}, \texttt{20qQBBB1}\}$ is a (2,3)-$\mathsf{deflate_slow}$ instance set of total length 43. Note that we used repeated bytes here for clarity; in practice, we usually build instance sets by appending random bytes in Σ. While unnecessary in practice, we can also verify that the bounds on $k, m, \mathcal{T}$ hold, namely $c_{1,v} + cc_{\mathsf{H}}(T_{k,m}, v) \leq \mathsf{max_chain} < c_{2,v}$.

A remaining parameter to decide on is n. Constraint (ii) requires that $\mathsf{max_match} = 258$ divide $|S| - |S_1| - |S_2| = 86(n-1)$, which is equivalent to $n = 3n' + 1$ for some $n' \in \mathbb{N}$.[104] We choose $n = 1$ here for simplicity, but n can be arbitrarily large. In practice, one needs to choose n based on the intended amplification effect as well as the maximum query length L_Q.

Now we can instantiate the main template using the instance set $\mathcal{T}$. The resulting query Q is

```
-10pqQAAA20qQAAA1PqQAAA0pqQBBB20qQBBB11PqQBBB
0pqQAAA1PqQAAA20qQAAA0pqQBBB1PqQBBB20qQBBB-20qQBBB1-secret=1
0pqQAAA20qQAAA1PqQAAA0pqQBBB20qQBBB11PqQBBB0pqQAAA1PqQAAA20qQAAA0pqQBBB1PqQBBB20qQBBB1
```

[104] Similar to our constructions in Section 7.2.5, this requirement can be ignored unless we wish the query to preserve the difference in pointer positions.

Finally, we can optionally check that the other requirements hold. For example, one can check that the substring 10p is indeed distinguished using Definition 13.

7.3.3 Small diverse double-colliding set

Suppose the attacker can only find a diverse double-colliding set $\{A, B\} \subset \Sigma^*$ of two strings. Generally, using the set $\{A, B\} \subset \Sigma^*$, the attacker can construct a consistent $(k, 2)$-deflate_slow instance set $\mathcal{T} = \{T_{i,j}\}_{i\in[k],j\in\{1,2\}}$ on Σ for any $k \in \mathbb{N}_+$, which we have shown in Theorem 11. For clarity, we denote $\mathcal{T}$ as $\{A_1, B_1, \ldots, A_k, B_k\}$, where for each $i \in [k]$, we let $A_i = T_{i,1}$ and $B_i = T_{i,2}$. By Definition 10, for every $i_1, i_2 \in [k]$, $A_{i_1}[1] \neq B_{i_2}[1]$, but it is allowed and often true in practice that $A_{i_1}[1] = A_{i_2}[1] = A[1]$ and $B_{i_1}[1] = B_{i_2}[1] = B[1]$.

We briefly discuss why we cannot simply use the consistent $(k, 2)$-deflate_slow instance set $\mathcal{T}$ to instantiate the main template we described above. Suppose we do so; then $S_1 = B_1\|A_1\|B_2\|A_2\|\cdots\|B_k\|A_k$ and $S_2 = A_1\|B_1\|A_2\|B_2\|\cdots\|A_k\|B_k$. If $ck[1] = g$, then the LZ77 pointer is initially at $S[1] = S_1[1]$, and one would expect that LZ77 outputs a back-reference for B_1 and moves the pointer to the first byte of A_1 in S_1. However, unless $A_2[1] \neq A_1[1]$, LZ77 actually finds a longer match ($B_1\|A_2[1]$ or longer) in $S_2[1 : |S_2| - 1]$ from one of the dictionaries, so LZ77 outputs a longer back-reference and moves the pointer to an unexpected position, after which the behaviour of LZ77 is out of our control.

Our solution is to slightly change the orders of the elements of $\mathcal{T}$ in S_1 and S_2, such that the same byte string $T \in \mathcal{T}$ is followed by A_{i_1} in S_1 and B_{i_2} in S_2 for some $i_1, i_2 \in [k]$. By definition, $A_{i_1}[1] \neq B_{i_2}[1]$, so T is followed by different bytes in S_1 and S_2, thereby preventing LZ77 from finding a longer match for T when $ck[1] = g$.

We present a template for this setting, which only differs from the template above in the construction of S_1 and S_2. For simplicity, we only specify the most important requirements; the ancillary requirements and analysis are very similar to the template above.

Construction. Our template requires a consistent $(k, 2)$-deflate_slow instance set $\mathcal{T}$ on Σ, where k is even. Let v denote the hash value at which $\mathcal{T}$ is double-colliding. The query Q is of the form $\mathsf{dict}_1\|\mathsf{filler}\|\mathsf{dict}_2\|P\|g\|S$. Here dict_1 is the empty string ε and dict_2 represents the sequence of two byte strings $(S[0]\|S_1\|S_2[1 : |S_2| - 1], B_k)$ (or vice versa),[105] the filler is the empty string ε, and the chain is $S = (S_1\|S_2)^n$, where $n \in \mathbb{N}_+$ and

$$S_1 = A_2\|A_1\|B_2\|B_1\|\cdots\|A_k\|A_{(k-1)}\|B_k\|B_{(k-1)}$$
$$S_2 = A_1\|A_2\|B_1\|B_2\|\cdots\|A_{(k-1)}\|A_k\|B_{(k-1)}\|B_k.$$

Requirements. We require that $S[0] = B_k[|B_k|]$ and $S[0 : \mathsf{min_match} - 1]$ be distinguished.

We also require that

$$c_{1,v} + cc_{\mathsf{H}}(B_k, v) \leq \mathsf{max_chain} < c_{2,v},$$

where $c_{1,v}$ and $c_{2,v}$ are defined in the same way as we did for the main template. Note that $c_{1,v} \geq 4k + 2$, $cc_{\mathsf{H}}(B_k, v) \geq 2$ and $c_{2,v} \geq 8k$. As a consequence, for the special case where the only collisions at v in $(S_1\|S_2)^*$ come from $T_{i,j}[1 : \mathsf{min_match}]$ and $T_{i,j}[2 : \mathsf{min_match} + 1]$ for every $i \in [k]$ and $j \in \{1, 2\}$, we require

$$\frac{\mathsf{max_chain}}{8} < k \leq \frac{\mathsf{max_chain}}{4} - 1.$$

In addition, we require that at least one of the two following constraints on the instance set $\mathcal{T}$ hold:

(i) There exists $\tau \in \mathbb{N}_+$ and $k_0, \ldots, k_\tau \in \mathbb{N}$, where $\tau < \frac{\mathsf{window_size}}{2\cdot\mathsf{max_match}}$ and $0 = k_0 < k_1 < \cdots < k_\tau = k$, such that for every $t \in [\tau]$, k_t is even,[106]

$$\sum_{i=k_{t-1}+1}^{k_t} (|A_i| + |B_i|) = \mathsf{max_match},$$

and if $t < \tau$,

$$B_{(k_t-1)}\left[\left|B_{(k_t-1)}\right|\right]\|A_{(k_t+2)}[1 : \mathsf{min_match} - 1]$$

[105] As with the main template, if $|S_1| + |S_2| \geq \mathsf{max_match}$, then the dictionary can instead just represent $S[0]\|S_1\|S_2$.

[106] We require k_t to be even mainly to simplify our description of the other requirements.

and

$$B_{k_t}\left[|B_{k_t}|\right] \| A_{(k_t+1)}\left[1 : \mathsf{min_match} - 1\right]$$

are distinguished. In addition, we require that $B_{(k-1)}\left[\left|B_{(k-1)}\right|\right] \| A_1\left[1 : \mathsf{min_match} - 1\right]$ be distinguished.

(ii) The integer $|S_1| + |S_2|$ divides $2 \cdot \mathsf{max_match}$ (or, equivalently, $|S_1| = |S_2|$ divides $\mathsf{max_match}$), $\mathsf{max_match}$ divides $|S| - |S_1| - |S_2|$, and, if $|S_1| + |S_2|$ does not divide $\mathsf{max_match}$, then we require that $B_{(k-1)}\left[\left|B_{(k-1)}\right|\right] \| A_1\left[1 : \mathsf{min_match} - 1\right]$ be distinguished.

Performance. The performance of this template is expected to be the same as the main template. In particular, If $ck[1] = g$, then the LZ77 output for $P\|g\|S$ consists of $kmn + 1$ back-references; otherwise, the LZ77 output only consists of $\lceil \ell_S / \mathsf{max_match} \rceil + 1$ back-references, and likely one literal for the last byte $S[\ell_S]$. However, in practice, constructing a suitable instance set $\mathcal{T}$ from only two byte strings may require appending more random bytes, so the average length of elements in $\mathcal{T}$ may be larger, making the query slightly less efficient.

7.3.4 No diverse double-colliding set

We now discuss the case where the attacker cannot find any diverse double-colliding set on Σ. It is still possible to perform collision-based amplification by multiplexing two or more double-colliding sets.

Suppose the attacker can find two singleton sets $\{A\}$ and $\{B\}$ that are double-colliding on Σ, such that $A[1] \neq B[1]$. By appending different random bytes to A and B, the attacker can generally construct a $(k, 2)$-$\mathsf{deflate_slow}$ instance set $\mathcal{T} = \{A_1, B_1, \ldots, A_k, B_k\}$ for any $k \in \mathbb{N}_+$; however, $\mathcal{T}$ cannot be consistent, as otherwise $\{A_1, B_1\}$ would be a diverse double-colliding set, contradictory to our assumption.

The construction of our template for this setting is exactly the same as the second template, except that our template requires a $(k, 2)$-$\mathsf{deflate_slow}$ instance set $\mathcal{T}$ on Σ, where k is even, such that $\mathcal{T}$ is *not* consistent. The performance is expected to be the same as the second template with the same parameters.

The requirements and analysis of this template are very similar to those of previous templates. Specifically, the main requirements of this template are the same as the second template, except for the bounds on $\mathsf{max_chain}$, which we give below.

Let v_1 (resp. v_2) be the hash value at which the set $\{A_i\}_{i\in[k]}$ (resp. $\{B_i\}_{i\in[k]}$) is double-colliding; by assumption, $v_1 \neq v_2$. We require that

$$c_{1,v_1} \leq \mathsf{max_chain} < c_{2,v_1}$$

and

$$c_{1,v_2} + cc_{\mathsf{H}}(B_k, v_2) \leq \mathsf{max_chain} < c_{2,v_2},$$

where c_{1,v_1}, c_{1,v_2}, c_{2,v_1} and c_{2,v_2} are defined in the same way as in the previous templates. As a consequence, for the special case where the only collisions at v_1 or v_2 in $(S_1\|S_2)^*$ come from $A_i\left[1 : \mathsf{min_match}\right]$, $A_i\left[2 : \mathsf{min_match} + 1\right]$, $B_i\left[1 : \mathsf{min_match}\right]$ and $B_i\left[2 : \mathsf{min_match} + 1\right]$ for every $i \in [k]$, we require

$$\frac{\mathsf{max_chain}}{4} < k \leq \frac{\mathsf{max_chain}}{2} - 2.$$

Therefore, S_1 and S_2 are generally longer, which may make this template infeasible for some atypical compressor configurations (e.g. if the sliding window size $\mathsf{window_size}$ is small) and attack scenarios (e.g. if the maximum query length L_Q is small).

7.3.5 No double-colliding set

Next, we briefly discuss one of the most unfortunate cases for the attacker, where the attacker can only find weakly double-colliding sets rather than double-colliding sets on Σ. Recall from Definition 8 that a set of k byte strings $\{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^{>\mathsf{min_match}}$ is weakly double-colliding at hash values $\{v_1, v_2\}$ with regard to H on Σ, if for every $i \in [k]$, $\{\mathsf{H}(T_i\left[1 : \mathsf{min_match}\right]), \mathsf{H}(T_i\left[2 : \mathsf{min_match} + 1\right])\} = \{v_1, v_2\}$.

For simplicity, we assume that the attacker can find a diverse weakly double-colliding set of at least $m \geq 2$ strings. By appending different random bytes, the attacker can then generally construct a consistent weak (k, m)-$\mathsf{deflate_slow}$ instance set on Σ for some $k \in \mathbb{N}_+$ as we define below, where we highlight the differences from the definition of a $\mathsf{deflate_slow}$ instance set (Definition 10) in brown.

Definition 14. *(consistent weak* deflate_slow *instance set) Let* $k, m, \mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$, $\Sigma \subseteq \{0, \ldots, 255\}$, *and let* H *be a DEFLATE hash function with the parameters* min_match *and* hash_bits. *A set of* km *byte strings* $\mathcal{T} = \{T_{1,1}, T_{1,2}, \ldots, T_{1,m}, \ldots, T_{k,1}, T_{k,2}, \ldots, T_{k,m}\} \subseteq \Sigma^*$ *is a consistent weak* (k, m)-deflate_slow *instance set with regard to* H *on* Σ, *if*

1. *For every* $(i_1, \ldots, i_m) \in [k]^m$, $\{T_{i_1,1}, \ldots, T_{i_m,m}\}$ *is diverse;*
2. $\mathcal{T}$ *is weakly double-colliding at some hash values* $\{v_1, v_2\}$*;*
3. *For every* $T_1, T_2, T_3, T_4 \in \mathcal{T}$, *where* $T_1 \neq T_3$ *or* $T_2[1] \neq T_4[1]$, *it holds that* $T_1\,[2 : |T_1|]\,\|T_2[1]$ *is not a substring of* $T_3 \| T_4$.

Note that any consistent (k, m)-deflate_slow instance set is also a consistent weak (k, m)-deflate_slow instance set, but not vice versa.

We can use a consistent weak (k, m)-deflate_slow instance set on Σ to instantiate the first or the second template, depending on whether $m \geq 3$. Intuitively, the analyses of the previous templates still hold if we modify the requirements on k, m and $\mathcal{T}$ to take into account that each string in $\mathcal{T}$ now roughly amounts to one collision instead of two. More specifically, let $\{v_1, v_2\}$ be the set of hash values at which the instance set $\mathcal{T}$ is weakly double-colliding. If $v_1 = v_2$, then $\mathcal{T}$ is also double-colliding, and each $T \in \mathcal{T}$ contributes to at least two entries in the hash chain corresponding to v_1 and v_2; otherwise, if $v_1 \neq v_2$, then each $T \in \mathcal{T}$ contributes to at least one entry in the hash chain corresponding to v_1, and similarly for the hash chain corresponding to v_2.

The main requirements are the same as we specified for the previous templates, except that we adjust the bounds on max_chain accordingly: we require that

$$c_{1,v_1} + cc_{\mathsf{H}}(T_{k,m}, v_1) \leq \mathsf{max_chain} < c_{2,v_1}$$

and

$$c_{1,v_2} + cc_{\mathsf{H}}(T_{k,m}, v_2) \leq \mathsf{max_chain} < c_{2,v_2},$$

where c_{1,v_1}, c_{1,v_2}, c_{2,v_1} and c_{2,v_2} are defined in the same way as in the previous templates. By definition, $cc_{\mathsf{H}}(T_{i,j}, v_1) \geq 1$ and $cc_{\mathsf{H}}(T_{i,j}, v_2) \geq 1$ for each $i \in [k]$ and $j \times [m]$, $c_{1,v_1} \geq km + 1$, $c_{1,v_2} \geq km + 1$, $c_{2,v_1} \geq 2km$, and $c_{2,v_2} \geq 2km$. As a consequence, for the special case where $v_1 \neq v_2$ and the only collisions at v_1 or v_2 in $(S_1 \| S_2)^*$ come from $T_{i,j}\,[1 : \mathsf{min_match}]$ and $T_{i,j}\,[2 : \mathsf{min_match} + 1]$ for every $i \in [k]$ and $j \in [m]$, we require

$$\frac{\mathsf{max_chain}}{2} < km \leq \mathsf{max_chain} - 2.$$

Similar to the case above where no diverse double-colliding set is known, these requirements imply that S_1 and S_2 are generally longer for queries constructed from weakly double-colliding sets, which may make these queries infeasible for some atypical compressor configurations.

7.3.6 Alternative instance set constraints

Finally, we sketch a slightly different way to instantiate the templates presented in Sections 7.3.2 to 7.3.5. Specifically, we give different constraints on the composition of the instance set $\mathcal{T}$, while keeping the construction and the other requirements unchanged.

Recall that we have introduced several advanced bootstrapping methods in Section 7.2.5, including aligning segments and aligning collisions. The template instantiations above can all be seen as formalising and extending the aligning segments method, because our previous constraints on the instance set $\mathcal{T}$ force the total length of $\mathcal{T}$ to either divide or be divisible by max_match. In comparison, template instantiations that satisfy the alternative instance set constraints below correspond to the aligning collisions method, where we force every element of the instance set $\mathcal{T}$ to be of the same carefully chosen length ℓ. While the alternative constraints are generally harder to meet, they allow the total length of the instance set to be relatively flexible, which can lead to more efficient queries in some cases and help bypass a restriction in the miniz compressor.

Constraints. For each of the templates in Sections 7.3.2 to 7.3.5, we still require a (k, m)-deflate_slow instance set $\mathcal{T}$ that is consistent or consistent weak if necessary and satisfies the bounds on k, m and $\mathcal{T}$ according to its respective requirements. Our new constraints on the instance set $\mathcal{T}$ are as follows. For $x, y \in \mathbb{N}_+$, let $\gcd(x, y)$ denote the greatest common divisor of x and y. Let $\ell \in \mathbb{N}_+$, where $\mathsf{min_match} + 2 \leq \ell < \min\{\mathsf{nice_length} + 2, \mathsf{max_match}\}$, $d = \gcd(\mathsf{max_match}, \ell)$, $d' = \gcd(\mathsf{max_match}, 2km\ell)$ and $\rho = (2km\ell \bmod \mathsf{max_match})$. Note that d divides d' and d' divides ρ by definition. We require that:

1. $|T_{i,j}| = \ell$ for every $(i,j) \in [k] \times [m]$;
2. $d \notin \{1,2\}$;
3. For every integer $0 < i \leq \frac{(2km-1)\ell}{d'}$, $(S_1 \| S_2)\,[id' : id' + \ell - 2]$ is distinguished;
4. For every integer $\frac{(2km-1)\ell}{d'} < i < \frac{2km\ell}{d'}$, $(S_1 \| S_2 \| S_1)\,[id' : id' + \ell - 2]$ appears exactly $n-1$ times in the query Q, and its hash value $\mathsf{H}((S_1 \| S_2 \| S_1)\,[id' : id' + \mathsf{min_match} - 1])$ is distinguished;
5. If $\rho \neq 0$, then $\rho \geq \mathsf{min_match}$ and $(S_1 \| S_2)\,[2km\ell - \rho + 1 : 2km\ell]$ does not appear in the sliding windows before $S[2km\ell - \rho]$ and $S[2km\ell - \rho + 1]$;
6. $\mathsf{max_match}/d'$ divides $n-1$.

We briefly explain the purpose of each constraint below.

- Constraint 1 ensures that, for each $\iota \in [|S|/\ell]$, $S\,[(\iota-1)\ell + 1 : \iota\ell] = T_{i,j}$ for some $(i,j) \in [k] \times [m]$. This constraint simplifies our analysis and can be further relaxed, which we do not discuss here.
- Constraint 2 implies $(t \cdot \mathsf{max_match} \bmod \ell) \notin \{1,2\}$ for every $t \in \mathbb{N}$ by Bézout's identity. It also implies $d' \geq d \geq 3$. Note that, if $\mathsf{max_match}$ is prime, then this constraint cannot hold with $\ell < \mathsf{max_match}$, thus excluding any meaningful amplification under this constraint.
- Constraint 3 ensures that LZ77 finds the intended sequence of back-references if the guess is incorrect, except for the corner cases excluded by constraints 4 and 5. Recall that $S\,[0 : \mathsf{min_match} - 1]$ is already required to be distinguished for every template above. We consider substrings of length $\ell - 1$ here to avoid the case where some substring is entirely contained within one element of the instance set $\mathcal{T}$ and therefore cannot be distinguished. To see why our choice of the length $\ell - 1$ works, note that $\{(id' - 1 \bmod \ell), (id' \bmod \ell), \ldots, (id' + \ell - 2 \bmod \ell)\} = \{0, \ldots, \ell - 1\}$, and that $(id' - 1 \bmod \ell) \notin \{0,1\}$ and $(id' \bmod \ell) \notin \{1,2\}$ by constraint 2.[107] It follows that there exists some integer $0 < j < 2km$ such that $j \cdot \ell + 1 \in \{id' + 1, \ldots, id' + \ell - 2\}$; that is, $(S_1 \| S_2)\,[id' : id' + \ell - 2]$ overlaps with $(S_1 \| S_2)\,[(j-1) \cdot \ell + 1 : j \cdot \ell]$ and $(S_1 \| S_2)\,[j \cdot \ell + 1 : (j+1) \cdot \ell]$, each being an element of the instance set $\mathcal{T}$ by constraint 1.
- Constraint 4 complements constraint 3 by restricting the LZ77 behaviour when the pointer is near the end of some $S_1 \| S_2$ in the chain S for the case where the guess is incorrect. The reason for the case distinction is that no substring of $S_2\,[(km-1)\ell + 1 : km\ell] = T_{k,m}$ can be a distinguished substring according to Definition 13, because $T_{k,m}$ appears at least $2n + 1$ times in the query Q; we address this issue in constraint 4 by adapting the definition of a distinguished substring.
- Constraint 5 excludes the corner case where $|S_1| + |S_2|$ is very close to some multiple of $\mathsf{max_match}$, such that the intended match for $S\,[2km\ell - \rho : 2km\ell - 1]$ cannot be found. Note that constraint 2 implies that either $\rho = 0$ or $\rho \geq 3$.
- Constraint 6 guarantees that the query keeps the difference in LZ77 pointer positions depending on whether the guess is correct or not, and this constraint can be ignored if the difference is not needed. While constraint 6 severely limits the choice of n, one can significantly relax this constraint by truncating the chain S; we do not discuss it here and refer interested readers to our implementation.

Analysis. We analyse the behaviour and output of LZ77 for $P \| g \| S$ in the query. For the case where $ck[1] = g$, our analysis in Section 7.3.2 still applies, since it does not rely on the constraints on the instance set $\mathcal{T}$. We analyse the case where $ck[1] \neq g$ below, assuming that LZ77 first finds a match for P in the sliding window and moves the pointer to $S[0] = g$ in the query Q.

First, we argue that LZ77 outputs $\tau := \lfloor (|S_1| + |S_2|)/\mathsf{max_match} \rfloor = \lfloor 2km\ell/\mathsf{max_match} \rfloor$ back-references of $\mathsf{max_match}$ bytes as well as at most one shorter back-reference, and moves the pointer to $S[|S_1| + |S_2|] = S[2km\ell]$. We discuss two possible cases depending on whether $\tau = 0$ or not.

- $\tau = 0$, i.e. $2km\ell < \mathsf{max_match}$. Since $S\,[0 : \mathsf{min_match} - 1]$ is distinguished by assumption, LZ77 finds a match $S[0] \| S_1 \| S_2\,[1 : |S_2| - 1]$ of $2km\ell$ bytes in one of the dictionaries. LZ77 cannot find a longer match in the sliding window by assumption 2 in Section 7.3.2. The match also cannot be improved by lazy matching, because LZ77 cannot find a match longer than ℓ bytes at $S[1]$, according to our analysis for the case where $ck[1] = g$. Therefore, LZ77 outputs a back-reference for $S[0] \| S_1 \| S_2\,[1 : |S_2| - 1]$ and moves the pointer to $S[2km\ell]$.
- $\tau > 0$, i.e. $2km\ell \geq \mathsf{max_match}$. Since $S\,[0 : \mathsf{min_match} - 1]$ is distinguished by assumption, LZ77 outputs a back-reference for $S\,[0 : \mathsf{max_match} - 1]$ of $\mathsf{max_match}$ bytes and moves the pointer to $S[\mathsf{max_match}]$. By constraint 3, $S\,[t \cdot \mathsf{max_match} : t \cdot \mathsf{max_match} + \ell - 2]$ is distinguished for every

[107] In more detail, if $(id' \bmod \ell) \in \{1,2\}$ for some integer i, then we have $\gcd(d', \ell) \in \{1,2\}$ by Bézout's identity. Since d divides both d' and ℓ by definition, it must be that d divides 1 or 2, which would violate constraint 2.

$t \in [\tau - 1]$, so we can continue the above process and deduce that LZ77 outputs τ back-references, $(S[(t-1) \cdot \mathsf{max_match} : t \cdot \mathsf{max_match} - 1])_{t \in [\tau]}$, and moves the pointer to $S[\tau \cdot \mathsf{max_match}]$. Note that $\tau \cdot \mathsf{max_match} = 2km\ell - \rho$ by definition.

If $\tau \cdot \mathsf{max_match} = 2km\ell$, then the pointer is already at $S[2km\ell]$. Otherwise, by constraint 5, we have $\tau \cdot \mathsf{max_match} + \mathsf{min_match} - 1 < 2km\ell$, we discuss two cases depending on whether $\rho \geq \ell$:

- If $\rho \geq \ell$, then by constraint 3, LZ77 finds a match for $S[\tau \cdot \mathsf{max_match} : 2km\ell - 1]$ in a dictionary, which by constraint 5 cannot be extended or improved by lazy matching.
- If $\rho < \ell$, then by constraint 5, LZ77 cannot find a match longer than $S[\tau \cdot \mathsf{max_match} : 2km\ell - 1]$ in the sliding window, while by constraint 4, LZ77 can find $S[\tau \cdot \mathsf{max_match} : 2km\ell - 1]$ in a dictionary. Therefore, LZ77 finds a match for $S[\tau \cdot \mathsf{max_match} : 2km\ell - 1]$, which by constraint 5 cannot be extended or improved by lazy matching.

In both cases, LZ77 outputs a back-reference $S[t \cdot \mathsf{max_match} : 2km\ell - 1]$ and moves the pointer to $S[2km\ell]$.

Let $\kappa = (|S| - |S_1| - |S_2|)/\mathsf{max_match} = 2(n-1)km\ell/\mathsf{max_match}$; by constraint 6, we have $\kappa \in \mathbb{N}$. At pointer position $S[2km\ell]$, since $S[2km\ell : 2km\ell + \mathsf{min_match} - 1] = S[0 : \mathsf{min_match} - 1]$ is distinguished, LZ77 outputs a back-reference for $S[2km\ell : 2km\ell + \mathsf{max_match} - 1]$ of $\mathsf{max_match}$ bytes and moves the pointer to $S[2km\ell + \mathsf{max_match}]$. The back-reference points to $S[0 : \mathsf{max_match} - 1]$. Repeating this process for κ times by applying constraint 3 or constraint 4 based on the pointer position, we deduce that LZ77 outputs κ back-references

$$(S[2km\ell + (t-1) \cdot \mathsf{max_match} : 2km\ell + t \cdot \mathsf{max_match} - 1])_{t \in [\kappa]}$$

and moves the pointer to $S[|S|]$.

In conclusion, LZ77 outputs

$$\texttt{<L}2km\ell\texttt{,D*>}\|(\texttt{<Lmax_match,D}2km\ell\texttt{>})^{\kappa}$$

if $2km\ell < \mathsf{max_match}$, and

$$(\texttt{<Lmax_match,D*>})^{\tau}\|\texttt{<L}\rho\texttt{,D*>}\|(\texttt{<Lmax_match,D}2km\ell\texttt{>})^{\kappa}$$

otherwise, where $\texttt{<L}\rho\texttt{,D*>}$ is not included in the LZ77 output if $\rho = 0$. In both cases, LZ77 moves the pointer to $S[|S|]$, the last byte of the chain S.

Application: miniz. Our alternative instance set constraints are especially useful for targeting the miniz compressor. Recall from Section 3.5 that LZ77 in miniz stops looking for a match as soon as having encountered a position in set $\{2^{15} \cdot i + 1\}_{i \in \mathbb{N}}$ on the hash chain and discards any match starting at such positions. As a consequence, instantiations of our templates for collision-based amplification in Section 7.3 are unlikely to work as desired if the chain S contains a substring $T_{i,j} \in \mathcal{T}$ that starts at such a position. This issue appears inevitable for indefinitely long queries unless we carefully choose the lengths of the byte strings in $\mathcal{T}$, which can be achieved using our alternative instance set constraints.

We give some additional conditions sufficient for achieving unlimited amplification against miniz using the templates in in Sections 7.3.2 to 7.3.5 with our alternative set constraints. Let $p_S \in \mathbb{N}_+$ denote the position of $S[1]$ in the plaintext. Crucially, the attacker in our model does not necessarily know the value of p_S, which depends on data unknown to the attacker before the query Q. Nevertheless, for small $\Delta \in \mathbb{N}$, the attacker can prepend a string of length Δ to the query Q to increase p_S by Δ. As we will see, this trick allows the attacker to randomise p_S to satisfy the additional conditions with a constant probability. Assume for simplicity that $p_S + \Delta + |S_1| + |S_2| \leq 2^{15}$ for any reasonably small Δ. We fix $\mathsf{max_match} = 258$ since this is hard-coded in miniz.[108] The additional conditions are as follows:

1. $\gcd(\mathsf{max_match}, 2^{15})$ does not divide $p_S + |S_1| + |S_2| - 1$. This condition ensures that the position of $S[t \cdot \mathsf{max_match} + |S_1| + |S_2|]$ is not in $\{2^{15} \cdot i + 1\}_{i \in \mathbb{N}}$ for every $t \in \mathbb{N}$, so that our analysis for the case where $ck[1] \neq g$ in Section 7.3.6 still holds.
2. $\gcd(\ell, 2^{15})$ does not divide p_S.

Condition 1 is equivalent to p_S being even, because $\gcd(\mathsf{max_match}, 2^{15}) = \gcd(258, 2^{15}) = 2$ and $|S_1| = |S_2|$. Assuming $\gcd(\mathsf{max_match}, \ell) \notin \{1, 2\}$, which is required by our alternative constraints, the smallest ℓ that satisfies both condition 1 and condition 2 for some p_S is $\ell = 12$. For $\ell = 12$, both

[108] https://github.com/richgel999/miniz/blob/3.1.1/miniz_tdef.h#L90

conditions are satisfied if and only if $p_S \bmod 4 = 2$. Through increasing p_S by Δ sampled uniformly at random from $\{0, 1, 2, 3\}$, the attacker can fulfil $p_S \bmod 4 = 2$ and thus fulfil the additional conditions with probability $1/4$ for arbitrary unknown p_S. A larger ℓ can increase the success probability but makes the query less efficient.

We note that it appears possible to design templates tailored to miniz that have better performance and potentially achieve unlimited amplification with high probability, which we leave for future work.

7.4 Templates for deflate_fast

For the last part of this section, we turn our focus to the deflate_fast compression strategy. The first two templates for deflate_slow (Sections 7.3.2 and 7.3.3) also work for deflate_fast under appropriate assumptions. For simplicity, we use LZ77 to denote the LZ77 algorithm with respect to some (unspecified) DEFLATE implementation and configuration. Unless otherwise specified, we assume that DEFLATE uses the deflate_fast strategy.

We recall the main differences between deflate_fast and deflate_slow and discuss their implications for collision-based amplification against deflate_fast:

1. LZ77 in deflate_fast does not perform lazy matching. As a result, it is not necessary, albeit possible, to use a double-colliding set or its variants for collision-based amplification against deflate_fast.
2. LZ77 in deflate_fast skips inserting all but the first position within a matched substring into the hash table if the match is strictly longer than max_insert bytes, whereas LZ77 in deflate_slow inserts every position in the sliding window into the hash table. This difference complicates collision-based amplification against deflate_fast, because the attacker has to additionally ensure that the matches they intend for LZ77 to discover have been inserted into the hash table beforehand.
3. Designed for compression speed, the deflate_fast strategy is usually associated with smaller parameters for compression, which poses different design challenges for collision-based amplification against deflate_fast. For example, in zlib, max_chain is 4, 8 or 32 in compression levels using deflate_fast, while it is 4096 in the highest compression level (see Table 2). As a result, there is little need to consider the case where max_chain is large, but one needs to pay attention to whether the template allows max_chain to be very small.

7.4.1 deflate_fast instance set

Analogous to the definition of a consistent deflate_slow instance set in Definition 10, we give the definition of a consistent deflate_fast instance set below, highlighting the differences from Definition 10 in brown.

Definition 15. *(consistent* deflate_fast *instance set) Let* $k, m, \mathsf{min_match}, \mathsf{hash_bits} \in \mathbb{N}_+$, $\Sigma \subseteq \{0, \ldots, 255\}$, *and let* H *be a DEFLATE hash function with the parameters* min_match *and* hash_bits. *A set of* km *byte strings* $\mathcal{T} = \{T_{1,1}, T_{1,2}, \ldots, T_{1,m}, \ldots, T_{k,1}, T_{k,2}, \ldots, T_{k,m}\} \subseteq \Sigma^*$ *is a consistent* (k, m)-deflate_fast *instance set with regard to* H *on* Σ, *if*

1. *For every* $(i_1, \ldots, i_m) \in [k]^m$, $\{T_{i_1,1}, \ldots, T_{i_m,m}\}$ *is diverse;*
2. $\mathcal{T}$ *is colliding at some hash value* v*;*
3. *For every* $T_1, T_2, T_3, T_4 \in \mathcal{T}$, *where* $T_1 \neq T_3$ *or* $T_2[1] \neq T_4[1]$, *it holds that* $T_1 \| T_2[1]$ *is not a substring of* $T_3 \| T_4$.[109]

Compared to Definition 10, we have weakened some conditions in Definition 15, because LZ77 in deflate_fast does not perform lazy matching. We remark that any consistent (k, m)-deflate_slow instance set is also a consistent (k, m)-deflate_fast instance set, but not vice versa.

In Definition 15, we do not consider the case where $\mathcal{T}$ is not consistent, i.e. $\{T_{i,j}\}_{i \in [k]}$ is colliding for every $j \in [m]$, but $\mathcal{T}$ is not colliding. The reason is that, similar to deflate_slow, finding a consistent (k, m)-deflate_fast instance set is generally as hard as finding a diverse colliding set of size m, which is easy for most parameter choices and DEFLATE hash functions as long as m is small.

7.4.2 Adapting templates to deflate_fast

For collision-based amplification against deflate_fast, we can use a consistent (k, m)-deflate_fast instance set on Σ to instantiate the template for the deflate_slow strategy in Section 7.3.2 or Section 7.3.3, depending on whether $m \geq 3$.[110] Assuming that the requirements that we state below also hold, the

[109] Similar to Definition 10, a weaker property would suffice here.

[110] Intuitively, for the deflate_slow strategy, we can also use a consistent (k, m)-deflate_fast instance set $\mathcal{T}$ to instantiate the first or the second template in Section 7.3 if $\min_{T \in \mathcal{T}} |T| \geq \mathsf{max_lazy}$, in which case lazy matching is essentially disabled for the query. This is useful for, e.g. compression level 4 in zlib.

performance of the constructed query is expected to be the same as what we previously claimed for deflate_slow. In particular, If $ck[1] = g$, then the LZ77 output for $P\|g\|S$ consists of $kmn + 1$ back-references; otherwise, the LZ77 output consists of $\lceil \ell_S/\mathsf{max_match} \rceil + 1$ back-references, and likely one literal for $S[\ell_S]$.

For deflate_fast, we give different bounds on max_chain. Let v be the hash value at which $\mathcal{T}$ is colliding, let $c_{1,v}$ and $c_{2,v}$ be defined in the same way as in Section 7.3.2, and let

$$c'_{2,v} = 2km + 2\sum_{i=1}^{k} \sum_{\substack{j\in[m] \\ |T_{i,j}|\le \mathsf{max_insert}}} cc_{\mathsf{H}}\left(T_{i,j}\left[2 : |T_{i,j}|\right], v\right).$$

By definition, $c_{2,v} \ge c'_{2,v} \ge 2km$. Generally, we require that

$$c_{1,v} + cc_{\mathsf{H}}(T_{k,m}, v) \le \mathsf{max_chain} < c'_{2,v},$$

where $c_{1,v} \ge km + 1$ and $cc_{\mathsf{H}}(T_{k,m}, v) \ge 1$ by definition. And for the special cases where

- $|T_{i,j}| > \mathsf{max_insert}$ for every $i \in [k]$ and $j \in [m]$ and $|T_{k,m}| > \mathsf{max_insert} + 1$, *or*
- the only collisions at v in $(S_1\|S_2)^*$ come from $T_{i,j}\left[1 : \mathsf{min_match}\right]$ for every $i \in [k]$ and $j \in [m]$,

we require

$$km + 2 \le \mathsf{max_chain} < 2km.$$

We also modify the constraints on the instance set $\mathcal{T}$ for the case where it simultaneously holds that $|S_1| + |S_2|$ divides $2 \cdot \mathsf{max_match}$, $|S_1| + |S_2|$ does not divide max_match and $|S_1| + |S_2| > \mathsf{max_insert}$. For this special case, we no longer require that max_match divide $|S| - |S_1| - |S_2|$ but instead require that max_match divide $|S| - |S_1| - 2|S_2| - \mathsf{max_match}$. This is to account for the fact that LZ77 does not insert $S[|S_1|]$ into the hash table, causing the output to differ when the guess is incorrect.[111]

To apply a similar analysis to Section 7.3.2, we have the following new requirements, stated informally:

1. The position of every substring $T \in \mathcal{T}$ in the dictionaries is inserted into the hash table.
2. In the definition of distinguished substrings (Definition 13), we additionally require that the position of the first occurrence of a distinguished substring Z in the query Q be inserted into the hash table.

Note that it is generally not necessary to check these requirements in practice.

A formal statement of the new requirements would appear quite verbose, because whether LZ77 inserts a position into the hash table or not may depend on the LZ77 behaviour at multiple previous positions, and because we do not assume any fixed constructions for the dictionaries and the filler. We omit the formal requirements here; nevertheless, we specify a sufficient condition under which a position in Q is inserted into the hash table, and one can repeat this condition to derive sufficient conditions for our new requirements.

Fix a position $Q[p]$ within the query Q; for simplicity, we assume that the position p satisfies $\mathsf{max_insert} \le p \le \ell_Q - \mathsf{max_insert}$ and that the LZ77 pointer falls within $Q\left[1 : p - 1\right]$ at some point.[112] We require that $Q\left[p - \lceil \mathsf{max_insert}/2 \rceil : p\right]$ and $Q\left[p : p + \lceil (\mathsf{max_insert} + 1)/2 \rceil\right]$ do not appear in the sliding window before $Q[\max\{1, p - \mathsf{max_match} + 1\}]$ and the sliding window before $Q[p]$.

We now show that the above condition ensures that $Q[p]$ is inserted into the hash table. Suppose, for a contradiction, that $Q[p]$ is not inserted into the hash table. This implies that LZ77 finds a match strictly longer than max_insert that overlaps with $Q[p]$. By assumption, the matched substring starts in $Q\left[1 : p - 1\right]$; denote the matched substring as $Q\left[a : b\right]$, where $1 \le a < p \le b$ (we allow $b \ge \ell_Q$ by abusing notation). By definition, $b - a + 1 \le \mathsf{max_match}$, so $a \ge p + 1 - \mathsf{max_match}$. If $a \le p - \lceil \mathsf{max_insert}/2 \rceil$, then $Q\left[p - \lceil \mathsf{max_insert}/2 \rceil : p\right]$ appears in the sliding window before $Q[\max\{1, p - \mathsf{max_match} + 1\}]$ or the sliding window before $Q[p]$. It follows that $a \ge p - \lceil \mathsf{max_insert}/2 \rceil + 1$. We obtain similarly that $b \le p + \lceil (\mathsf{max_insert} + 1)/2 \rceil - 1$. Therefore, $b - a + 1 \le \lceil \mathsf{max_insert}/2 \rceil + \lceil (\mathsf{max_insert} + 1)/2 \rceil - 1 = \mathsf{max_insert}$, contradictory to our assumption that the match $Q\left[a : b\right]$ is strictly longer than max_insert.

We also briefly discuss adapting our alternative instance set constraints in Section 7.3.6 to deflate_fast. On the one hand, one can remove the constraint $\gcd(\ell, \mathsf{max_match}) \ne 2$ and therefore choose a smaller

[111] Let $\kappa = (|S| - |S_1| - 2|S_2|)/\mathsf{max_match} \in \mathbb{N}^+$; for $P\|g\|S$ in Q, if the guess is incorrect, then LZ77 is expected to output `<L`ℓ_P`,D*>`$\|$`<L(`$|S_1| + |S_2|$`),D*>`$\|$`<Lmax_match,D(`$|S_1| + |S_2|$`)>`$\|$`<L`$|S_2|$`,D*>`$\|$`(<Lmax_match,D*>)`$^{\kappa-1}$ and move the pointer to $S[\ell_S]$.

[112] These assumptions are generally irrelevant in practice and can be achieved by prepending a small amount of filler data or an align gadget (Section 8.2.1) to the query Q.

ℓ in order to build more efficient queries. On the other hand, the fact that LZ77 does not insert certain positions into the hash table makes it considerably more nuanced to determine the new constraints. While the alternative construction for deflate_slow also works for deflate_fast in practice, we omit a discussion here for brevity; note that max_insert is not used in miniz (Section 3.5).

7.4.3 max_chain = 4

Unfortunately, since $km \geq 3$, the bounds on max_chain cannot be satisfied when max_chain ≤ 4, whereas we have max_chain $= 4$ at compression level 1 in zlib. Our solution for this case is to slightly modify the main template to also add each string in $\mathcal{T}$ into one of the dictionaries, which can be seen as reversing the improvement we performed in our examples of "a more efficient extension" in Section 7.2.3. More specifically, instead of representing the sequence of byte strings $(S[0] \| S_1 \| S_2\,[1 : |S_2| - 1]\,, T_{k,m})$, dict_2 (or dict_1) now represents

$$(S[0] \| S_1 \| S_2\,[1 : |S_2| - 1]\,, T_{1,1}, \ldots, T_{1,m}, \ldots, T_{k,1}, \ldots, T_{k,m}).$$

Note that, if $k > 1$, then it is generally desirable to use more than one separator to construct the dictionaries, so as to prevent LZ77 from finding a match longer than max_insert starting at a separator and consequently failing to insert the first index of some dictionary entry $T \in \mathcal{T}$ into the hash table. We refrain from a detailed discussion as we are primarily interested in the case where $k = 1$.

For the modified construction, we require that $c_{1,v} \leq \mathsf{max_chain} < c'_{2,v}$ or, for the special cases above, $km + 1 \leq \mathsf{max_chain} < 2km$. These bounds can be satisfied when max_chain $= 4$ with $k = 1$, $m = 3$ and a suitable $\mathcal{T}$. The compressed length difference is expected to be the same as we stated previously; however, the query becomes longer by around $\sum_{T \in \mathcal{T}} (|T| + 1) \geq km(\mathsf{min_match} + 1)$ bytes (assuming one-byte separators are used to construct the dictionaries), making the query slightly less efficient, especially if n is small.

We remark that one can use the same method to loosen the bounds on max_chain for the deflate_slow templates to allow max_chain $\in \{8, 9\}$; this, however, is of little practical interest, as max_chain is at least 16 in deflate_slow for all DEFLATE implementations we surveyed.

7.4.4 max_chain = 1

Recall from Section 3.5 that the deflate_quick strategy used at level 1 in zlib-ng can be seen as deflate_fast with max_chain $= 1$ and max_insert $<$ min_match. It appears hard to adapt our existing templates to this case, because LZ77 stops looking for a match as soon as it encounters a single collision. However, when max_chain is very small, it is no longer necessary to build collision-based amplification queries using colliding sets, as one can achieve the same effect by carefully repeating the same non-colliding strings. We sketch below a general construction for collision-based amplification when max_chain $= 1$, which can be further optimised if max_match is even.

Construction. Let $k, m \in \mathbb{N}_+$, where $k \geq$ min_match and $km \leq$ (max_match $-$ min_match)$/4$. Let R be a byte string such that $|R| =$ max_match $- 4km$ and $S[0] = R[|R|]$. Let A, B be two byte strings of length k such that $\{A[1], B[1], R[1]\} = 3$, i.e. A, B and R each start with a different byte. In practice, A, B, R can be constructed using random bytes in Σ subject to the above requirements.

The query Q is of the form $\mathsf{dict}_1 \| \mathsf{filler} \| \mathsf{dict}_2 \| P \| g \| S$. Here dict_1 is the empty string ε and dict_2 represents the sequence of two byte strings $(S[0] \| S_1, S_1[|S_1|] \| S_2)$ (or vice versa), the filler is the empty string ε, and the chain is $S = (S_1 \| S_2)^n$, where $n \in \mathbb{N}_+$ and

$$S_1 = (A \| A \| B \| B)^m \| R$$
$$S_2 = (B \| B \| A \| A)^m \| R.$$

Note that $|S_1| = |S_2| =$ max_match by definition, and n can be arbitrarily large.

If $ck[1] = g$, then LZ77 likely encodes $P \| g \| S$ in the query as

$$\texttt{<L}(\ell_P + 1)\texttt{,D*>} \| \left((\texttt{<L}k\texttt{,D*>})^{4m} \| \texttt{<L}|R|\texttt{,D*>} \right)^{2n}.$$

Otherwise, if $ck[1] \neq g$, then LZ77 likely encodes $\mathsf{gadgets} = P \| g \| S$ in the query as

$$\texttt{<L}\ell_P\texttt{,D*>} \| (\texttt{<Lmax_match,Dmax_match>})^{2n} \| S[|S|].$$

We do not give a formal analysis here, but readers can easily verify that no substring A (resp. B and R) in the query Q and its subsequent occurrence in Q are followed by the same byte. For example, the first and the third A in $(A\|A\|B\|B)^2$ are followed by $A[1]$, while the second and the fourth A are followed by $B[1]$, where $A[1] \neq B[1]$ by definition. Therefore, when the pointer is at the first byte of A, B or R in the chain S, LZ77 likely only finds a match for A, B or R but not longer matches.

This construction is generally more efficient in practice than the other collision-based amplification constructions, because A and B can be of $\mathsf{min_match}$ bytes, while the byte strings within an instance set are typically longer; as a result, when considering queries of similar lengths, the LZ77 output length difference is typically larger for queries built using this construction. It is therefore tempting to construct collision-based amplification queries using repeated strings when $\mathsf{max_chain} > 1$. We refrain from further discussion but remark that this is clearly feasible if $\mathsf{max_chain}$ is small.

8 CRIME automata

In this section, we introduce a new paradigm for constructing queries for compression side-channel attacks against DEFLATE. We construct queries to the compression length oracle as *CRIME automata*, which turn the LZ77 algorithm into a simple state machine that can be programmed in a certain sense. CRIME automata give the attacker precise control over the behaviour of LZ77 and the oracle responses, and the attacker can construct complex CRIME automata modularly by chaining simpler building blocks called *gadgets*. We describe and analyse a variety of gadgets for CRIME automata, and we show that our amplification techniques in Sections 6 and 7 can also be seen as special gadgets. As an interesting application, the attacker can use CRIME automata to efficiently merge multiple queries into a single one, thereby decreasing the number of queries needed for secret recovery in compression side-channel attacks.

8.1 Overview

Motivation. A core observation that motivates CRIME automata is that one can construct queries modularly from smaller gadgets, each having a specific functionality. As the readers may have noticed in Sections 6 and 7, most of our amplification techniques build queries of the form $\mathsf{dict}_1\|\mathsf{filler}\|\mathsf{dict}_2\|\mathsf{gadgets}$, where $\mathsf{gadgets} = P\|g\|S$. Essentially, the string of gadgets $\mathsf{gadgets}$ consists of two parts: the first part $P\|g$ checks whether the guess g is equal to $ck[1]$, and the second part S amplifies the compressed length difference by making LZ77 find very different back-references. It is therefore natural to treat each part as an individual gadget that provides a certain functionality. In this case, the two gadgets are a match gadget (Section 8.3.1) and an amplify (or reverse-amplify) gadget (Section 8.4).

Another core observation is that we can see LZ77 pointer positions as *states*, and abuse the LZ77 algorithm in DEFLATE in order to control state transitions. This observation allows us to design gadgets precisely and make them work in concert to perform a complex functionality. We take a closer look at the behaviour of LZ77 on the two gadgets in the above example. Assume that the LZ77 pointer is initially at the first byte of the match gadget $P\|g$; if $ck[1] = g$, then LZ77 is expected to output a back-reference for $P\|g$ and move the pointer to the first byte of S, right after the gadget; otherwise, LZ77 is expected to output a back-reference for P and move the pointer to g, the last byte of the gadget. That is, the match gadget creates a one-byte difference in the LZ77 pointer position depending on whether $ck[1] = g$. The amplify gadget S compresses to a larger or smaller size depending on the initial position of the LZ77 pointer, rather than the actual content of the previous gadget.

CRIME automaton. A CRIME automaton, or an automaton in short, is a substring of a query that is separated from other parts of the query, designed to achieve a certain functionality. A query can contain multiple CRIME automata, concatenated together with optional separators, in which case the automata jointly influence the response from the compression length oracle. For simplicity, we will equate a CRIME automaton with a query unless stated otherwise.

Structure. CRIME automata have the same structure as the query structure we described in Section 6.1. Recall that a query Q contains up to four parts: two dictionary strings dict_1 and dict_2, each representing a sequence 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 gadgets, each being a crafted byte string that performs a specific functionality.

Execution. We refer to querying the compression length oracle with the automaton Q as an execution.

State. During an execution, the state of an automaton is indicated by the LZ77 pointer position as LZ77 scans through gadgets. Without loss of generality, we denote the state where the LZ77 pointer is at the first byte of a gadget as true, and we denote the state where the LZ77 pointer is at the last byte of a gadget as false. All other pointer positions correspond to undefined states. The state of an automaton is well-defined if every gadget has at least two bytes.

We note that it is clearly possible but likely impractical in general to use ternary or quaternary states (in addition to the undefined state) for a CRIME automaton.

Output. The output of an automaton Q from an execution is the response to the query Q given by the compression length oracle, i.e. the compressed length of the plaintext with Q embedded after the secret ck. The attacker can typically learn information about the secret ck from the output, such as whether the first byte of the secret $ck[1]$ is equal to a guess g. Note that, even though CRIME automata are intended to be deterministic, different executions of the same automaton Q may yield different outputs, because noise in the plaintext or randomness in the compression algorithm (e.g. random padding) may affect the output.

We stress that the output of an automaton is *not* its final state from an execution.[113] While appending an appropriate amplify gadget to the existing gadgets allows the attacker to infer the final state of the automaton from the output, it is not necessary (but usually desirable) to program an automaton in that way. Furthermore, as we will see in Section 8.6, for some automata, the attacker can learn more information from the output than the final state.

Gadget. We denote a gadget by a byte string S of length $\ell_S \geq 2$. Similar as in Sections 6 and 7, we abuse notation to let $S[0]$ denote the byte immediately before S in the plaintext, and for each $i \in \mathbb{N}_+$, we let $S[\ell_S + i]$ denote the i-th byte located after S in the plaintext, or a special end symbol if there are fewer than i bytes after S.[114]

A gadget takes an initial state and transitions it into another state, possibly creating a significant difference in the LZ77 output for different initial states or different sliding window contents. Formally speaking, the initial state of a gadget is the state when the LZ77 pointer falls within the gadget or at the byte before the gadget (i.e. $S[0 : \ell_S]$) for the first time. The initial state of a gadget can thus be true, false, or undefined. With the exception of the align gadget (Section 8.2.1), we assume that the initial state of a gadget is either true or false, and that the behaviour of LZ77 is unspecified when the initial state of a gadget is undefined. The state transition derives the initial state of the next gadget, if there is any.

Note that, while every gadget in an automaton directly influences its output, most gadgets are too short to substantially influence the output by itself. Therefore, for simplicity, we do not analyse the compressed lengths of the short gadgets, including all except telescoped-match gadgets and amplify gadgets. That said, when building an automaton with many gadgets, one should ensure that the effects of telescoped-match gadgets and amplify gadgets on the output outweigh those of other gadgets.

A main part of this section is devoted to defining various gadgets, categorised as follows:

- **Separation gadgets** (Section 8.2). These gadgets are used to separate gadgets from other parts of the query, or to separate two gadgets. The *align gadget* (Section 8.2.1) transitions any initial state to true, directing the LZ77 pointer to the first byte of the next gadget, and the *NOP gadget* (Section 8.2.2) does not change the initial state if it is true or false.
- **Match gadgets** (Section 8.3). These gadgets formalise and extend on the idea of guessing a secret byte or string by observing matches. The *match gadget* (Section 8.3.1) checks whether a target string can be found in the sliding window, the *unmatch gadget* (Section 8.3.2) checks whether a target string cannot be found in the sliding window, and the *telescoped-match gadget* (Section 8.3.3) is a reformulation of the telescoping technique (Section 6.2).
- **Amplify gadgets** (Section 8.4). This type of gadget is a reformulation of the chaining technique (Section 6.3) and collision-based amplification (Section 7). Depending on the direction of amplification, we can subdivide amplify gadgets into *forward-amplify gadgets* and *reverse-amplify gadgets*.

[113] We informally define the final state of an automaton as follows: the final state of an automaton Q from an execution is false if the LZ77 pointer falls at the last byte of the automaton Q at some point, and true otherwise.

[114] Our constructions of a gadget S may use $S[0]$, the byte before the gadget, but they do not use any byte after the gadget, thus avoiding circularity when building a CRIME automaton. However, we may use the bytes after the gadget for analyses of correctness, so the behaviour of LZ77 at a gadget may depend on its preceding and subsequent gadgets.

– **Logical gadgets** (Section 8.5). These gadgets transition the initial state to another state based on the initial state and possibly on whether a string can be found in the sliding window. The *NOT gadget* (Section 8.5.1) inverts the initial state, the *AND-match gadget* (Section 8.5.2) sets the state to true if and only if the initial state is true and a target string is found in the sliding window, and the *OR-match gadget* (Section 8.5.3) sets the state to false if and only if the initial state is false and a target string is *not* found in the sliding window.

There may be more than one construction for the same gadget. Our list of gadgets is not exhaustive, and one can certainly design new gadgets for different functionalities.

8.2 Separation gadgets

We start by describing two simple separation gadgets, the align gadget and the NOP gaget. They have no concrete functionalities per se, but are rather used to separate gadgets from other parts of the query, or to separate two gadgets, in order to reduce interference and decrease the error probability.

8.2.1 Align gadget

An align gadget forces the LZ77 pointer that falls within the gadget to move just past it, to the first byte 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 main motivation for align gadgets is that the initial state of the first gadget in an automaton is generally unknown and may be affected by other parts of the plaintext, so it is desirable to first set the state of the automaton to a known value.[115] Additionally, in some cases, the initial state of a gadget must be true for it to perform its intended functionality.

In our example setting, suppose the query Q is preceded by `query=` in the plaintext, and suppose the plaintext before the query contains a substring `memory=secret`. The plaintext is then

$$\cdots \| \texttt{secret=} \| ck \| \cdots \| \texttt{memory=secret} \| \cdots \| \texttt{query=} \| Q \| \cdots$$

An attacker may set the query Q to be `secret=1` to check if $ck[1]$ is equal to `1`. However, for the substring `query=secret=1`, LZ77 likely finds a match `ry=secret` in `memory=secret` and outputs `que<L9,D*>=1` regardless of whether the guess is correct, causing the attack to fail. To solve this problem, the attacker can prepend an align gadget to `secret=1` in the query to separate it from `query=`, such that LZ77 moves the pointer to the first byte of `secret=1` at some point, and LZ77 likely outputs a back-reference for `secret=1` if $ck[1] = \texttt{1}$.

The align gadget may not be necessary for real-world compression side-channel attacks, which generally allow some level of inaccuracy. Nevertheless, we note that, in the compression side-channel attack against Threema [PST23, Appendix B], the former of the two canary strings required for the attack possibly acted as an align gadget.

We describe two constructions for the align gadget, and provide a formal analysis for the first one; other constructions are also possible. Care must be taken when using multiple align gadgets to avoid interference among them.

Repeated bytes: construction. The repetition of some byte that ideally does not appear in the sliding window, such as `@@@@@@`, works as an align gadget. For example, LZ77 likely outputs `@<L5,D1>` for `@@@@@@` and moves the pointer just past it if the LZ77 pointer is initially at the first byte of the gadget.

More formally, the gadget S is of the form c^{ℓ_S} for some $c \in \Sigma$, such that:

1. For some $k \in [\mathsf{min_match}]$, the byte string c^k does not appear in the sliding window before the gadget, and $\ell_S \geq \mathsf{min_match} + k + 1$.
2. The gadget is not followed by the byte c in the plaintext, i.e. $S[\ell_S + 1] \neq c$.

The first condition can be relaxed in practice, especially for certain compressor configurations.

We illustrate the behaviour of LZ77 in our example in Fig. 5.

[115] A similar concept is the NOP slide in buffer overflow attacks.

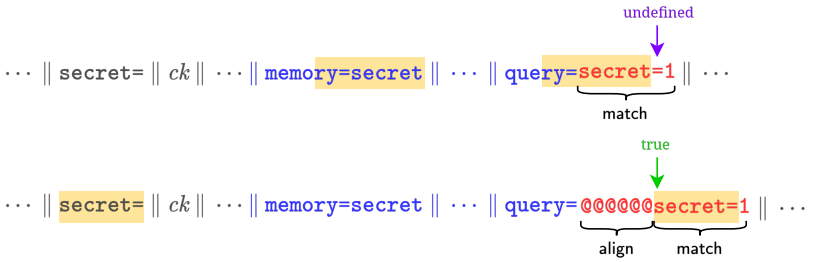


Fig. 5: Illustration of the align gadget in our example setting. The query Q is in red, the likely LZ77 matches are highlighted, and the LZ77 pointer is shown as an arrow that indicates the final state of the gadget. For simplicity, we did not highlight the overlapping LZ77 match (`@@@@@`) within the align gadget.

Repeated bytes: analysis. We now analyse the behaviour of LZ77 at the gadget S. Let $i_0 \in [\ell_S]$ be the index of the LZ77 pointer when it falls within S for the first time; that is, the LZ77 pointer is initially at $S[i_0]$. The index i_0 is well-defined and $i_0 \le k$, because otherwise LZ77 would have found a match in the sliding window that contains c^k as a substring, contradictory to condition 1. At position $S[i_0]$, LZ77 has two possible behaviours:

- If $i_0 > 1$ and LZ77 has inserted the position of $S[i_0 - 1]$ into the hash table, then LZ77 finds a match $S[i_0 : \ell_S]$ of length $\ell_S - i_0 + 1 \ge \mathsf{min_match} + 2$, starting at $S[i_0 - 1]$. Suppose, for a contradiction, that the match $S[i_0 : \ell_S]$ can be extended or improved by lazy matching. This would imply that $S[i_0 + 1 : \ell_S] \| S[\ell_S + 1]$ appears in either S or the sliding window before S (categorised according to the position of the first byte); the former case contradicts condition 1, and the latter case contradicts condition 2. Therefore, LZ77 outputs a back-reference `<L`$(\ell_S - i_0 + 1)$`,D1>` and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.
- Otherwise, LZ77 has not inserted any position of S into the hash table. By condition 1, LZ77 inserts the current position into the hash table, outputs $S[i_0]$ as a literal, and moves the pointer to $S[i_0 + 1]$. At $S[i_0 + 1]$, by a similar argument as above, LZ77 outputs a back-reference `<L`$(\ell_S - i_0)$`,D1>` for $S[i_0 + 1 : \ell_S]$, and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.

Known match. For the deflate_slow strategy, one can use a known substring in the sliding window as an align gadget.[116] In our example, the attacker may use `memory` or `secret` as an align gadget. If no suitable known substring is available, then the attacker may sample a random string as the gadget and add it to one of the dictionaries.

We only give an informal justification. Suppose the LZ77 pointer first falls at $S[i_0]$ in the gadget, where $i_0 \in [\ell_S]$ is small enough such that $\ell_S - i_0 + 1 \ge \mathsf{min_match}$.[117] LZ77 likely finds a match $S[i_0 : \ell_S]$ from the known substring in the sliding window, which, under suitable assumptions, cannot be extended or improved by lazy matching.[118] Therefore, LZ77 outputs a back-reference for $S[i_0 : \ell_S]$ and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.

Misalign gadget. Besides setting the state of the automaton to true using an align gadget, one can alternatively set the state to false using a "misalign" gadget. A misalign gadget forces the LZ77 pointer to move to the last byte of the gadget, transitioning any state to false. However, this gadget is mainly of theoretical interest, as no gadget in our work requires its initial state to be false.

Appending one byte to an align gadget immediately yields a misalign gadget, unless the added byte violates the requirements for the align gadget. Readers will see that concatenating an align gadget with a NOT gadget (Section 8.5.1) also achieves the same effect as a misalign gadget does, albeit in a more complicated way.

Alternatively, one can construct a misalign gadget using a random byte string. The intuition is that LZ77 likely does not find any back-reference that starts from a position within the gadget, so LZ77 will gradually move the pointer to the last byte of the gadget. We omit a detailed analysis here.

[116] This construction also works for the deflate_fast strategy if $\ell_S \le \mathsf{max_insert}$.
[117] We enforce this by requiring that $S[0 : k]$ does not appear in the sliding window for some small $k \in [\ell_S]$.
[118] In our example, the known substring `memory=` does not work as an align gadget, because it forms a longer match `memory=secret` with the next gadget.

8.2.2 NOP gadget

Some gadgets have special requirements on the preceding byte or the subsequent bytes of the gadget to function correctly, which may not always be satisfied by the automaton. In addition, in some cases, concatenating two gadgets creates a substring across the two gadgets that affects the correctness of subsequent gadgets. To solve these problems, one can use a NOP gadget to separate two interfering gadgets. A NOP gadget acts as a separator by transitioning an initial state of true to true, and an initial state of false to false. Note that behaviour of the NOP gadget is unspecified if its initial state is undefined.

Construction. To construct a NOP gadget, one can sample the gadget S as a random string of length $\mathsf{min_match} \leq \ell_S \leq \mathsf{max_match}$, and add both $S[0 : \ell_S - 1]$ and S to dict_1 or dict_2. One can also choose the gadget S as a known substring of length $\mathsf{min_match} \leq \ell_S \leq \mathsf{max_match}$ in the sliding window, and add $S[0 : \ell_S - 1]$ to dict_1 or dict_2.[119]

In our example setting, a NOP gadget between a match gadget `secret=1` and an align gadget `111111` can be a random string `HJx0c6`, and we add `1HJx0c` and `HJx0c6` to a dictionary. The query can be

```
-1HJx0c-HJx0c6-secret=1HJx0c6111111
```

where we use `-` as separators in dict_2 and set dict_1 and filler to ε.

We illustrate the behaviour of LZ77 in our example in Fig. 6.

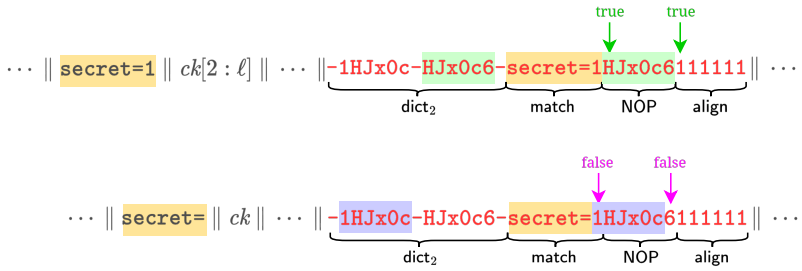


Fig. 6: Illustration of the NOP gadget in our example setting. The query Q is in red, the likely LZ77 matches are highlighted, and the LZ77 pointer is shown as an arrow that indicates the state, with two arrows representing the initial and the final state of the gadget. It shows the effect when $ck[1] = \texttt{1}$ (top) and when $ck[1] \neq \texttt{1}$ (bottom).

If the initial state of the NOP gadget is true, which is expected to happen if $ck[1] = \texttt{1}$, then LZ77 likely outputs `<L6,D15>` for `HJx0c6` and moves the pointer to the state true; otherwise, if the initial state of the NOP gadget is false, which is expected to happen if $ck[1] \neq \texttt{1}$, then LZ77 likely outputs `<L6,D21>` for `1HJx0c` and moves the pointer to the state false.

Requirements. We require that:

1. LZ77 can find $S[0 : \ell_S - 1]$ and S in the sliding window before the gadget.
2. The three substrings $S[0 : \ell_S]$, $S[1 : \ell_S + 1]$ and $S[2 : \ell_S + 2]$ do not appear in the sliding window.

These requirements are likely satisfied by a random string that is not too short, but one needs to pay attention to whether the requirements hold when using a known substring as a NOP gadget. In our example setting, `secret=` is generally not a suitable candidate for a NOP gadget if it is followed by the byte `1` in the automaton; the reason is that, if $ck[1] = \texttt{1}$ and the initial state is true, LZ77 likely outputs a back-reference for the longer match `secret=1` and moves the pointer to an undefined state.

Analysis. We now analyse the behaviour of LZ77 in detail.

– The initial state is true; i.e. the LZ77 pointer is initially at $S[1]$. By condition 1, LZ77 finds a match for S in the sliding window. By condition 2, LZ77 cannot find a longer match at $S[1]$ or improve the match by lazy matching. Therefore, LZ77 outputs a back-reference for S and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.

[119] As we will see in Section 8.4, our construction can be seen as a degenerate case of an amplify gadget constructed using distance-based chaining (Section 6.3.1) when $m = 1$.

– The initial state is false; i.e. the LZ77 pointer is initially at $S[0]$. By condition 1, LZ77 finds a match for $S[0 : \ell_S - 1]$ in the sliding window. By condition 2, LZ77 cannot find a longer match at $S[0]$ or improve the match by lazy matching. Therefore, LZ77 outputs a back-reference for $S[0 : \ell_S - 1]$ and moves the pointer to $S[\ell_S]$, corresponding to the state false.

8.3 Match gadgets

A match gadget sets the state of the automaton based on whether LZ77 finds a match for a target string in the sliding window. This can be used by the attacker to obtain information about the secret. Recall that the attacker cannot directly observe the state of the automaton; instead, the attacker can infer the match result by appending an amplify gadget that creates a visible difference in the output for different states, or by using a telescoped-match gadget, a variant of the match gadget that also amplifies the compressed length difference.

Virtually all compression side-channel attacks utilise queries containing what are in effect match gadgets, most commonly $P\|g$, the concatenation of the prefix and a guess for the first unknown byte. The intuition is that LZ77 replaces substrings that appear in the sliding window with back-references, and, more generally, a more redundant string is likely to have a shorter compressed length irrespective of the compressor. However, the role of the LZ77 pointer has gone unnoticed in prior work. The LZ77 pointer affects the behaviour of the match gadget, and the match gadget in turn determines the subsequent position of the LZ77 pointer. As we remarked in Section 8.1, the state change from the match gadget makes it possible to amplify the compressed length difference.

We formalise the method of exploiting LZ77 to detect whether a target string can be found in the sliding window as a match gadget. We also describe two variants of the match gadget: the *unmatch gadget* tests whether a target string *cannot* be found by LZ77 in the sliding window, and the *telescoped-match gadget* is a reformulation of the telescoping technique in Section 6.2.

All of our constructions for the match gadget assume that the initial state of the gadget is true, which can be ensured by prepending an align gadget. Therefore, we will only specify the behaviour of the match gadgets when the initial state is true. That said, the match gadgets we construct may still achieve the intended functionalities in practice even if the initial state is false or undefined.

8.3.1 Match gadget

A match gadget S checks whether the string S is present in the sliding window. If the initial state is true, then the match gadget sets the state to true if S can be found by LZ77 in the sliding window, and sets the state to false otherwise.

Construction. The gadget S is simply the target string of length $\mathsf{min_match} + 1 \leq \ell_S \leq \mathsf{max_match}$. We add the substring $S[1 : \ell_S - 1]$ to one of the dictionaries, unless LZ77 can already find the substring $S[1 : \ell_S - 1]$ in the sliding window.[120]

For example, to test whether $ck[1] = g$ in our attack model, the attacker can use the match gadget $S = P\|g$ to check whether the target string $P\|g$ appears in the sliding window. If we already know that P is in the sliding window, there is usually no need to add P to a dictionary. In our example setting, a query with a match gadget for the byte string `secret=1` is simply `secret=1`. If the attacker does not know that `secret=` can be found by LZ77 in the sliding window, then the query may be `-secret=-secret=1`, where we use `-` as the separator in dict_2 and omit dict_1 and filler. We illustrate this in Fig. 7.

If $ck[1] = \texttt{1}$, then LZ77 likely outputs a back-reference `<L8,D*>` for `secret=1` and terminates at the state true; otherwise, LZ77 likely outputs `<L7,D*>` for `secret=` and moves the pointer to `1`, corresponding to the state false.

Requirements. To ensure the correctness of the match gadget, we require that LZ77 can find $S[1 : \ell_S - 1]$ in the sliding window, but $S[2 : \ell_S + 1]$ does not appear in the sliding window.

Analysis. It is easy to see why the match gadget works correctly. By assumption, the initial state of the gadget is true; i.e. the LZ77 pointer is initially at $S[1]$.

[120] For simplicity, we require that $\ell_S > \mathsf{min_match}$, but in fact the match gadget still works correctly under suitable assumptions when $\ell_S = \mathsf{min_match}$, in which case we do not add $S[1 : \ell_S - 1]$ to a dictionary.

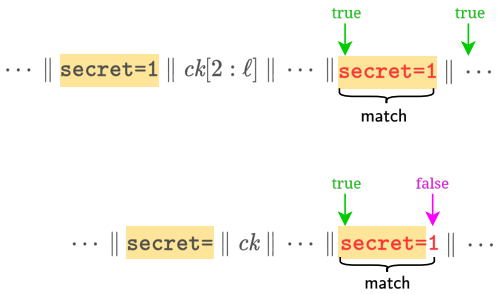


Fig. 7: Illustration of the match gadget in our example setting. The query Q is in red, the likely LZ77 matches are highlighted, and the LZ77 pointer is shown as an arrow that indicates the state, with two arrows representing the initial and the final state of the gadget. It shows the effect when $ck[1] = \texttt{1}$ (top) and when $ck[1] \neq \texttt{1}$ (bottom).

– If LZ77 finds S in the sliding window, then LZ77 finds a match for the entire gadget S, which by our requirements cannot be extended or improved by lazy matching. Therefore, LZ77 outputs a back-reference for S and moves its pointer to $S[\ell_S + 1]$, corresponding to the state true.
– Otherwise, if LZ77 does not find S in the sliding window, by our requirements, LZ77 finds a match $S\,[1 : \ell_S - 1]$ in the sliding window, which cannot be extended or improved by lazy matching. Therefore, LZ77 outputs a back-reference for $S\,[1 : \ell_S - 1]$ and moves the pointer to $S[\ell_S]$, corresponding to the state false.

8.3.2 Unmatch gadget

An unmatch gadget implements the opposite functionality of a match gadget. Namely, an unmatch gadget checks whether a target string T *cannot* be found by LZ77 in the sliding window. If the initial state is true, then the unmatch gadget sets the state to false if T can be found in the sliding window, and sets the state to true otherwise.

One can construct an unmatch gadget by combining a match gadget with a NOT gadget (Section 8.5.1). Here, we give a simpler construction exploiting lazy matching for the deflate_slow compression strategy.

Construction. Assume the compressor uses the deflate_slow compression strategy. The unmatch gadget is $S = T\|c$ of length $\ell_S = |T| + 1$, where T is a target string of length min_match $\leq |T| \leq$ max_lazy, and c is a one-byte string. We also add $S\,[2 : \ell_S] = T\,[2 : |T|]\,\|c$ to one of the two dictionaries, unless LZ77 can already find $S\,[2 : \ell_S]$ in the sliding window.

In our example setting, to probe whether `secret=1` is in the sliding window, we can use an unmatch gadget `secret=1/`, which adds `ecret=1/` to a dictionary. The query can then be `-ecret=1/-secret=1/`, where we use `-` as the separator in dict_2 and omit dict_1 and filler. If $ck[1] = \texttt{1}$, then LZ77 likely outputs a back-reference `<L8,D*>` for `secret=1` and moves the pointer to `/`, corresponding to the state false. Otherwise, for `secret=1/`, LZ77 likely first finds a match `secret=` and then improves it with the match `ecret=1/` found during lazy matching. As a result, LZ77 likely outputs `s<L9,D*>` and moves the pointer to the byte following `/`, corresponding to the state true.

Requirements. In addition to requiring that lazy matching is enabled and min_match $\leq |T| \leq$ max_lazy as stated above, we require that:

1. LZ77 can find $S\,[2 : \ell_S]$ in the sliding window.
2. The three substrings S, $S\,[2 : \ell_S + 1]$ and $S\,[3 : \ell_S + 2]$ do not appear in the sliding window.

One can choose c as a byte that rarely appears in the sliding window to increase the probability that condition 2 holds.

Analysis. By assumption, the initial state of the gadget is true; i.e. the LZ77 pointer is initially at $S[1]$. We discuss three possible cases:

– LZ77 finds T in the sliding window. By condition 2, the match T cannot be extended or improved by lazy matching. Therefore, LZ77 outputs a back-reference for $T = S[1 : \ell_S - 1]$ and moves the pointer to $S[\ell_S]$, corresponding to the state false.
– LZ77 finds a match for a strict prefix of T in the sliding window. The length of the match is strictly smaller than $|T| \leq$ max_lazy, so LZ77 performs lazy matching and, by condition 1, finds a match for $S[2 : \ell_S]$ in the sliding window. By condition 2, the match $S[2 : \ell_S]$ cannot be extended or improved by lazy matching. Therefore, LZ77 outputs $S[1]$ and back-reference for $S[2 : \ell_S]$ and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.
– LZ77 does not find any match for substrings starting at $S[1]$. LZ77 outputs the literal $S[1]$ and moves the pointer to $S[2]$, where, by condition 1, LZ77 finds a match for $S[2 : \ell_S]$ at $S[2]$. By condition 2, the match $S[2 : \ell_S]$ cannot be extended or improved by lazy matching, so LZ77 outputs a back-reference for $S[2 : \ell_S]$ and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.

8.3.3 Telescoped-match gadget

The telescoping technique in Section 6.2 can be seen as constructing a telescoped-match gadget that checks whether a target string of the form $P\|g$ can be found in the sliding window. Here P and g can be different than the prefix and guess we defined in our attack model. Based on whether the target string can be found in the sliding window, the telescoped-match gadget not only changes the state but also creates a usually measurable compressed length difference. If the initial state is true, then the telescoped-match gadget sets the state to true and decreases the output if the target string $P\|g$ can be found by LZ77 in the sliding window, and it sets the state to false and increases the output otherwise.

The construction, requirements and analysis of the telescoped-match gadget can be found in Section 6.2. Note that, similar to our construction of the match gadget, if LZ77 might not find P in the sliding window, then we also add P to one of the two dictionaries.

8.4 Amplify gadgets

Most gadgets work by introducing directed state changes. However, the state of the automaton is not visible from the output. To infer the state of the automaton at a given point, the attacker can insert an amplify gadget to turn the state difference into a measurable difference in the output. Amplify gadgets induce differences in the compressed length of the plaintext between the case where the initial state is true and the initial state is false, and keep the initial state unchanged.

We have implicitly presented several constructions for amplify gadgets when we introduced the chaining techniques in Section 6.3 and collision-based amplification in Section 7. The constructions, requirements and analyses for amplify gadgets can be easily extracted from our descriptions in Sections 6 and 7; in particular, the amplify gadget is the same as the amplification chain S in our techniques.

Depending on the "direction" of amplification, we can subdivide amplify gadgets into two types: the forward-amplify gadget decreases the compressed length when the initial state is true, and the reverse-amplify gadget increases the compressed length when the initial state is true.[121] For example, our constructions for collision-based amplification can all be seen as reverse-amplify gadgets.

We remark that an amplify gadget also provides the functionality of a NOP gadget. Conversely, our construction of the NOP gadget in Section 8.2.2 can be seen as a degenerate case of an amplify gadget constructed using distance-based chaining (Section 6.3.1).

8.5 Logical gadgets

The gadgets we have introduced so far do not give the attacker significantly more power than the techniques we described in Sections 6 and 7, as these gadgets can only form queries that one can imitate to a large extent without using the CRIME automata paradigm. In this section, we will introduce some logical gadgets that enable more control over states, unleashing the full power of CRIME automata.

8.5.1 NOT gadget

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

We give four constructions for NOT gadgets and formally analyse the first two constructions.

[121] Note that one can simulate one type of amplify gadget using another type of amplify gadget by prepending and appending a NOT gadget to the amplify gadget (Section 8.5.1).

Short NOT. We can construct a NOT gadget S as a short byte string of length $\ell_S = \mathsf{min_match} - 1$, and we add $S[0 : \ell_S]$ to a dictionary unless LZ77 can already find it in the sliding window.

In our example setting, to invert the state after the match gadget `secret=1`, the attacker can use a NOT gadget `!0`, and add `1!0` to the dictionary dict_2. We illustrate this in Fig. 8.

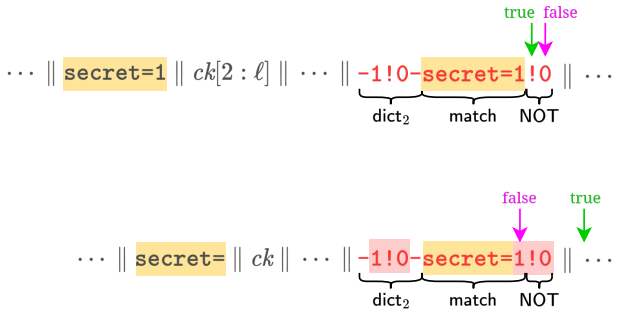


Fig. 8: Illustration of the short NOT gadget in our example setting. The query Q is in red, the likely LZ77 matches are highlighted, and the LZ77 pointer is shown as an arrow that indicates the state, with two arrows representing the initial and the final state of the gadget. It shows the effect when $ck[1] = \texttt{1}$ (top) and when $ck[1] \neq \texttt{1}$ (bottom).

If the initial state of the NOT gadget is true, then LZ77 likely does not find a match at `!` and advances the pointer by one byte to `0`, corresponding to the state false. Otherwise, if the initial state is false, then LZ77 likely finds a match `1!0` at the initial pointer position `1` and moves the pointer to the byte following `0`, corresponding to the state true.

In general, we require that $\mathsf{min_match} \geq 3$,[122] and:

1. LZ77 can find $S[0 : \ell_S]$ in the sliding window.
2. For every $i \in [\ell_S - 1]$, the substring $S[i : \ell_S + i]$ of length $\mathsf{min_match}$ does not appear in the sliding window.

The gadget S ideally contains one or more special bytes reserved for NOT gadgets, and random strings and known matches likely also work.

We now analyse the behaviour of LZ77 in detail.

- The initial state is true; i.e. the LZ77 pointer is initially at $S[1]$. By condition 2, LZ77 cannot find any match at $S[1]$, so LZ77 outputs the literal $S[1]$ and moves the pointer to $S[2]$. By repeating the same argument $\ell_S - 1$ times, we conclude that LZ77 outputs literals $S[1], \ldots, S[\ell_S - 1]$ and moves the pointer to $S[\ell_S]$, corresponding to the state false.
- The initial state is false; i.e. the LZ77 pointer is initially at $S[0]$. By condition 1, LZ77 finds a match for $S[0 : \ell_S]$ in the sliding window. By condition 2, the substring $S[1 : \ell_S + 1]$ does not appear in the sliding window, which means LZ77 cannot extend the match for $S[0 : \ell_S]$ or improve it using lazy matching. Therefore, LZ77 outputs a back-reference for $S[0 : \ell_S]$ and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.

Long NOT. It is not always possible to construct short NOT gadgets as we introduced above, especially when $\mathsf{min_match}$ is small or several NOT gadgets are used in the automaton. We give another construction of a NOT gadget, which is more versatile but simultaneously more complex.

Let k be an integer chosen by the attacker such that $\mathsf{min_match} \leq k \leq \mathsf{max_match}$; if $\mathsf{min_match} = 2$, then we require that $k = \mathsf{min_match} = 2$. The gadget S is a byte string of length $\ell_S = k + \mathsf{min_match} - 1$, and we add $S[0 : \mathsf{min_match} - 1]$, $S[\min\{2, \mathsf{min_match} - 1\} : \ell_S - 1]$ and $S[\mathsf{min_match} : \ell_S]$ to one of the dictionaries.[123]

In our example setting (with $\mathsf{deflate_fast}$), to invert the state after the match gadget `secret=1`, the attacker can use a NOT gadget `A!CDEF` with $k = 4$, and add `1A!`, `!CDE` and `CDEF` to the dictionary dict_2. If

[122] The construction still works when $\mathsf{min_match} = 2$, but it violates our assumption that a gadget has at least two bytes, which leads to (temporary) ambiguities in the state of the automaton.

[123] The length of $S[\min\{2, \mathsf{min_match} - 1\} : \ell_S - 1]$ is 3 if $\mathsf{min_match} = 2$, and k if $\mathsf{min_match} > 2$.

the initial state of the NOT gadget is true, then LZ77 likely does not find any match at `A` and advances the pointer to `!`, where LZ77 likely finds a match `!CDE` and moves the pointer to `F`, corresponding to the state false. Otherwise, if the initial state is false, then LZ77 likely outputs two back-references `<L3,D*><L4,D*>` for `1A!` and `CDEF` and moves the pointer to the byte following `F`, corresponding to the state true.

For our construction to work correctly, in addition to the requirements on k and ℓ_S, we require that:

1. LZ77 can find $S[0 : \mathsf{min_match} - 1]$, $S[\min\{2, \mathsf{min_match} - 1\} : \ell_S - 1]$ and $S[\mathsf{min_match} : \ell_S]$ in the sliding window.[124]
2. If $\mathsf{min_match} = 2$, then the substrings $S[0 : 2]$, $S[1 : 3]$ and $S[2 : 4]$ do not appear in the sliding window.
3. If $\mathsf{min_match} > 2$, then the substrings $S[1 : \mathsf{min_match}]$, $S[3 : \ell_S]$ and $S[\mathsf{min_match} + 1 : \ell_S + 1]$ do not appear in the sliding window.

Random strings likely satisfy the above requirements unless the alphabet Σ is too small. It is also desirable to include one or more unique bytes in $S[1 : \mathsf{min_match} - 1]$.

We now analyse the behaviour of LZ77 in detail.

- The initial state is true; i.e. the LZ77 pointer is initially at $S[1]$.
  - If $\mathsf{min_match} = 2$, then by condition 1, LZ77 finds a match $S[1 : \ell_S - 1] = S[1 : 2]$ at $S[1]$. By condition 2, LZ77 cannot extend the match or improve it using lazy matching. Therefore, LZ77 outputs a back-reference for $S[1 : \ell_S - 1]$ and moves the pointer to $S[\ell_S]$, corresponding to the state false.
  - Otherwise, if $\mathsf{min_match} > 2$, then by condition 3, LZ77 cannot find any match at $S[1]$, so LZ77 outputs the literal $S[1]$ and moves the pointer to $S[2]$. By condition 1, LZ77 finds a match $S[2 : \ell_S - 1]$ at $S[2]$, which cannot be extended or improved by lazy matching by condition 3. Therefore, LZ77 outputs a back-reference for $S[2 : \ell_S - 1]$ and moves the pointer to $S[\ell_S]$, corresponding to the state false.
- The initial state is false; i.e. the LZ77 pointer is initially at $S[0]$.
  - If $\mathsf{min_match} = 2$, then by condition 1, LZ77 finds a match $S[0 : \mathsf{min_match} - 1] = S[0 : 1]$ at $S[0]$. By condition 2, LZ77 cannot extend the match or improve it using lazy matching. Therefore, LZ77 outputs a back-reference for $S[0 : \mathsf{min_match} - 1]$ and moves the pointer to $S[\mathsf{min_match}]$. There, by condition 1, LZ77 finds a match for $S[\mathsf{min_match} : \ell_S] = S[2 : 3]$, which by condition 2 cannot be extended or improved using lazy matching. LZ77 therefore outputs a back-reference for $S[\mathsf{min_match} : \ell_S]$ and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.
  - Otherwise, if $\mathsf{min_match} > 2$, then by condition 1, LZ77 finds a match for $S[0 : \mathsf{min_match} - 1]$ at $S[0]$. By condition 3, LZ77 cannot extend the match or improve it using lazy matching, so LZ77 outputs a back-reference for $S[0 : \mathsf{min_match} - 1]$ and moves the pointer to $S[\mathsf{min_match}]$. At $S[\mathsf{min_match}]$, by condition 1, LZ77 finds a match for $S[\mathsf{min_match} : \ell_S]$ of length k, which cannot be extended or improved using lazy matching by condition 3. Therefore, LZ77 outputs a back-reference for $S[\mathsf{min_match} : \ell_S]$ and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.

A small issue of our long NOT construction is that it still relies on the presence or absence of certain short strings in the sliding window, such as $S[0 : \mathsf{min_match} - 1]$ and $S[1 : \mathsf{min_match}]$. As a result, the attacker cannot arbitrarily decrease the error probability by simply using random strings with larger k. We describe two alternative constructions that avoid this issue, targeting deflate_fast and deflate_slow respectively.

Fast NOT. Assume the compressor uses the deflate_fast compression strategy. Let k be an integer chosen by the attacker such that $\mathsf{min_match} \le k \le 1 + \mathsf{max_match}/2$. The gadget S is a byte string of length $\ell_S = 2k - 1$, and we add $S[0 : k - 1]$, $S[1 : \ell_S - 1]$ and $S[k : \ell_S]$ to one of the dictionaries.

In our example setting using deflate_fast, a NOT gadget after the match gadget `secret=1` can be `ABCDEFG` with $k = 4$, and we add `1ABC`, `ABCDEF` and `DEFG` to the dictionary dict_2. If the initial state of the NOT gadget is true, then LZ77 likely outputs a back-reference `<L6,D*>` for `ABCDEF` and advances the pointer to `G`, corresponding to the state false. Otherwise, if the initial state is false, then LZ77 likely outputs two back-references `<L4,D*><L4,D*>` for `1ABC` and `DEFG` and moves the pointer to the byte following `G`, corresponding to the state true.

[124] If $\mathsf{min_match} = 2$, then $k = 2$ and $\ell_S = 3$, and condition 1 stipulates that LZ77 can find $S[0 : 1]$, $S[1 : 2]$ and $S[2 : 3]$ in the sliding window.

This construction likely does not work correctly under deflate_slow when $k <$ max_lazy. In the above example, if the initial state is false and lazy matching is enabled, then LZ77 likely discards the initial match 1ABC in favour of a longer match ABCDEF from lazy matching, and moves the pointer to G, corresponding to the state false. However, it works correctly under deflate_slow if $k \geq$ max_lazy.[125]

Lazy NOT. Assume the compressor uses the deflate_slow compression strategy. Let k be an integer chosen by the attacker such that min_match $+ 1 \leq k \leq$ max_lazy. The gadget S is a byte string of length $\ell_S = \max\{2k-2, k+2\}$,[126] and we add $S[0:k-1]$, $S[2:\ell_S-1]$ and $S[k:\ell_S]$ (of length k, $\max\{2k-4, k\}$ and $\max\{k-1, 3\}$ respectively) to one of the dictionaries.

In our example setting using deflate_slow, a NOT gadget after the match gadget secret=1 can be ABCDEF with $k = 4$, and we add 1ABC, BCDE and DEF to the dictionary dict_2. If the initial state of the NOT gadget is true, then LZ77 likely first finds a match ABC within the dictionary entry 1ABC, and subsequently improves it with the match BCDE from lazy matching. Therefore, LZ77 likely outputs A<L4,D*> for ABCDE and advances the pointer to F, corresponding to the state false. Otherwise, if the initial state is false, then LZ77 likely outputs two back-references <L4,D*><L3,D*> for 1ABC and DEF and moves the pointer to the byte following F, corresponding to the state true.

8.5.2 AND-match

An AND-match gadget sets the state of the automaton as the logical conjunction of the initial state of the gadget and the state given by a match gadget for a target string T. If the initial state is true, then the AND-match gadget sets the state to true if LZ77 can find T in the sliding window, and to false otherwise. If the initial state is false, then the AND-match gadget sets the state to false, irrespective of whether LZ77 can find T in the sliding window.

Our construction of an AND-match gadget is conceptually simple and extremely close to our construction of a match gadget. The intuition is to force LZ77 to find a long match and move the pointer directly to the last byte of the gadget if the initial state is false.

Construction. The gadget S is simply the target string T of length min_match$+1 \leq |T| \leq$ max_match, and we add $S[0:\ell_S-1]$ to one of the dictionaries.

In our example setting, after a match gadget secret=1, an AND-match gadget for the target string admin=1 can simply be admin=1, and we add 1admin= into a dictionary, e.g. dict_2. The query can be -1admin=-secret=1admin=1, where we use - as the separator in dict_2 and omit dict_1 and filler. Note that one still needs to append an amplify gadget to make the state visible, which we omit here for simplicity.

If the initial state of the AND-match gadget is true, which is expected to happen if $ck[1] = 1$, then the behaviour of the AND-match gadget likely depends on whether LZ77 can find admin=1 in the sliding window. Either LZ77 outputs a back-reference <L7,D*> for admin=1 and moves the pointer to the byte following the gadget, corresponding to the state true, or LZ77 outputs a back-reference <L6,D*> for admin= and moves the pointer to 1, corresponding to the state false. Otherwise, if the initial state is false, which is expected to happen if $ck[1] \neq 1$, then LZ77 likely outputs a back-reference <L7,D*> for 1admin= and moves the pointer to the byte after the gadget, corresponding to the state false.

The concatenation of the previous gadget and the AND-match gadget sometimes creates unwanted interference, in which case we can prepend a NOP gadget (Section 8.2.2) to the AND-match gadget. In the above example, we can insert a NOP gadget ABCD between the two gadgets; the query then becomes

```
-1ABC-ABCD-Dadmin=-secret=1ABCDadmin=1
```

Requirements. In addition to requiring min_match $+ 1 \leq \ell_S \leq$ max_match, we require that:

1. LZ77 can find $S[0:\ell_S-1]$ and $S[1:\ell_S-1]$ in the sliding window.
2. The substrings $S[0:\ell_S]$ and $S[2:\ell_S+1]$ do not appear in the sliding window.

[125] The intuition is that every match we intend for LZ77 to find is of length at least $k \geq$ max_lazy, so lazy matching is essentially disabled for the NOT gadget.

[126] Equivalently, $\ell_S = 2k-2$ if $k \geq 4$, and $\ell_S = k+2$ otherwise.

Analysis. We now analyse the behaviour of LZ77 in detail.

- The initial state is true; i.e. the LZ77 pointer is initially at $S[1]$. We discuss two possible cases depending on whether LZ77 can find the target string $T = S$ in the sliding window.
  - If LZ77 finds S in the sliding window, then LZ77 finds a match for the entire gadget S, which by condition 2 cannot be extended or improved by lazy matching. Therefore, LZ77 outputs a back-reference for S and moves its pointer to $S[\ell_S + 1]$, corresponding to the state true.
  - Otherwise, if LZ77 does not find S in the sliding window, by condition 1, LZ77 finds a match $S[1 : \ell_S - 1]$ in the sliding window that cannot be extended. By condition 2, LZ77 also cannot improve the match $S[1 : \ell_S - 1]$ using lazy matching. Therefore, LZ77 outputs a back-reference for $S[1 : \ell_S - 1]$ and moves the pointer to $S[\ell_S]$, corresponding to the state false.
- The initial state is false; i.e. the LZ77 pointer is initially at $S[0]$. By condition 1, LZ77 finds a match for $S[0 : \ell_S - 1]$ in the sliding window. By condition 2, LZ77 cannot extend the match $S[0 : \ell_S - 1]$. Similarly, improving $S[0 : \ell_S - 1]$ using lazy matching would imply the presence of $S[1 : \ell_S + 1]$ in the sliding window, contradictory to condition 2. Therefore, LZ77 outputs a back-reference for $S[0 : \ell_S - 1]$ and moves the pointer to $S[\ell_S]$, corresponding to the state false.

8.5.3 OR-match

An OR-match gadget sets the state of the automaton as the logical disjunction of the initial state of the gadget and the state given by a match gadget for a target string T. If the initial state is true, then the OR-match gadget sets the state to true, irrespective of whether LZ77 can find T in the sliding window. If the initial state is false, then the OR-match gadget sets the state to true if LZ77 can find T in the sliding window, and to false otherwise.

Our construction of an OR-match gadget is more complex than that of an AND-match gadget. In particular, we cannot simply use the target string T as the OR-match gadget; instead, we construct the OR-match gadget by prepending a small "glue" string R to the target string T. The intuition is that, if the initial state is true, then we force the LZ77 pointer to first move to the middle of T and then move to the byte after T; otherwise, we force the LZ77 pointer to move to the first byte of T, and from this point on LZ77 behaves like a match gadget for T.

The OR-match gadget should be used with care, because our construction adds more than one non-random entry to the dictionaries, which may interfere with other gadgets.

Construction. Let T be the target string, where $\mathsf{min_match} + 1 \leq |T| \leq \mathsf{max_match}$. Let k be an integer chosen by the attacker such that

$$\mathsf{min_match} \leq k \leq |T|.$$

The OR-match gadget S we construct for the target string T is a byte string $S = R\|T$ of length $\ell_S = |R| + |T|$. Here R is a byte string chosen by the attacker such that

$$\mathsf{min_match} \leq |R| + 1 \leq \mathsf{max_match}$$

and

$$\mathsf{min_match} \leq |R| + |T| - k \leq \mathsf{max_match}.$$

We add three byte strings

$$\begin{aligned} S[1 : \ell_S - k] &= R\|T[1 : |T| - k] \\ S[0 : |R|] &= S[0]\|R \\ S[\ell_S - k + 1 : \ell_S] &= T[|T| - k + 1 : |T|] \end{aligned}$$

of length $|R| + |T| - k$, $|R| + 1$ and k respectively to one of the two dictionaries.[127] We also add $S[|R| + 1 : \ell_S - 1] = T[1 : |T| - 1]$ to one of the dictionaries, unless LZ77 can already find $T[1 : |T| - 1]$ in the sliding window.

[127] Note that, if the compressor uses the deflate_fast strategy, then changing the order of the three byte strings in the query may cause unwanted effects for our default dictionary constructions. For example, $S[1 : \ell_S - k]$ and $S[\ell_S - k + 1 : \ell_S]$ may form $S[1 : \ell_S - k]\|\mathsf{sep}_{\mathsf{dict}}\|S[\ell_S - k + 1 : \ell_S]$ in the dictionary, which contains $T[1 : |T| - k]\|\mathsf{sep}_{\mathsf{dict}}\|T[|T| - k + 1 : |T|]$ as a substring across the two entries. This substring may cause LZ77 not to insert certain entries in the dictionary into the hash table intended for other OR-match gadgets.

In our example setting, after the match gadget `secret=1`, an OR-match gadget for the target string `secret=2` can be `ORsecret=2`, and we add three byte strings `ORs`, `1OR` and `ecret=2` into dict_2; here the attacker chooses $k = 7$ and $R = \texttt{OR}$. The query (omitting the amplify gadget) can be

`-ORs-1OR-ecret=2-secret=1ORsecret=2`

where we use `-` as the separator in dict_2 and omit dict_1 and filler.

We illustrate this in Fig. 9.

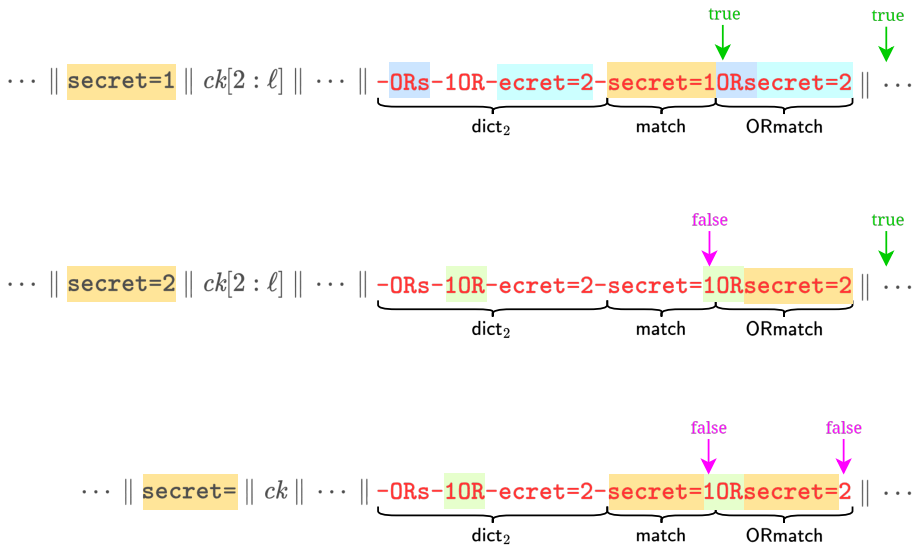


Fig. 9: Illustration of the OR-match gadget in our example setting. The query Q is in red, the likely LZ77 matches are highlighted, and the LZ77 pointer is shown as an arrow that indicates the state, with two arrows representing the initial and the final state of the gadget. It shows the effect when $ck[1] = \texttt{1}$ (top), $ck[1] = \texttt{2}$ (middle) and $ck[1] \neq \texttt{1}, ck[1] \neq \texttt{2}$ (bottom).

We discuss the likely behaviours of LZ77:

- If the initial state of the OR-match gadget is `true`, which is expected to happen if $ck[1] = \texttt{1}$, then LZ77 likely finds matches `ORs` and `ecret=2`, outputs `<L3,D*><L7,D*>` for `ORsecret=2`, and moves the pointer to the byte following the last byte `2`, corresponding to the state `true`.
- If the initial state of the OR-match gadget is `false`, which is expected to happen if $ck[1] \neq \texttt{1}$, then the behaviour of LZ77 depends on the presence of the target string in the sliding window, which in turn depends on whether $ck[1] = \texttt{2}$:
  - If LZ77 can find the target string `secret=2` in the sliding window, then LZ77 likely finds matches `1OR` and `secret=2`, outputs `<L3,D*><L8,D*>` for `1ORsecret=2`, and moves the pointer to the byte following `2`, corresponding to the state `true`.
  - Otherwise, LZ77 likely finds matches `1OR` and `secret=`, outputs `<L3,D*><L7,D*>` for `1ORsecret=`, and moves the pointer to `2`, corresponding to the state `false`.

Requirements. In addition to the above requirements on k, $|R|$ and $|T|$, we require that:

1. If lazy matching is enabled, then $|R| \geq \mathsf{max_lazy} - 1$ or $k = |T| - 1$.
2. LZ77 can find $S[0 : |R|]$, $S[1 : \ell_S - k]$, $S[|R| + 1 : \ell_S - 1]$ and $S[\ell_S - k + 1 : \ell_S]$ in the sliding window.
3. The substring $S[0 : |R| + 1]$ does not appear in the sliding window.
4. The substrings $S[1 : \ell_S - k + 1]$ and $S[2 : \ell_S - k + 2]$ do not appear in the sliding window.
5. The substring $S[\ell_S - k + 2 : \ell_S + 1]$ does not appear in the sliding window.

Readers can verify the consistency of our requirements using $k \geq \mathsf{min_match}$, $\ell_S - k \geq \mathsf{min_match}$ and a simple corollary $\ell_S - k \geq |R|$. It is also easy to see that the attacker-chosen parameters $k = |T| - 1$ and $|R| = \mathsf{min_match} - 1$ do not violate our requirements on k, $|R|$ and $|T|$, including condition 1; however, it is sometimes necessary to adjust these parameters in practice to ensure the other requirements hold and avoid interference with other gadgets. Note that condition 5 requires the absence of $T[|T| - k + 2 : |T|] \| S[\ell_S + 1]$ in the sliding window, which usually means k should not be too small.

Analysis. We now analyse the behaviour of LZ77 in detail.

- The initial state is true; i.e. the LZ77 pointer is initially at $S[1] = R[1]$. By condition 2, LZ77 finds a match for $S[1 : \ell_S - k]$ of length $\ell_S - k$ in the sliding window, which by condition 4 cannot be extended or improved using lazy matching. Therefore, LZ77 outputs a back-reference for $S[1 : \ell_S - k]$ and moves the pointer to $S[\ell_S - k + 1] = T[|T| - k + 1]$. By condition 2, LZ77 finds a match for $S[\ell_S - k + 1 : \ell_S]$ of length k in the sliding window, and by condition 5, LZ77 cannot extend the match or improve it using lazy matching. Therefore, LZ77 outputs a back-reference for $S[\ell_S - k + 1 : \ell_S]$ and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.
- The initial state is false; i.e. the LZ77 pointer is initially at $S[0]$. By condition 2, LZ77 finds a match for $S[0 : |R|]$ of length $|R| + 1$ in the sliding window, which cannot be extended by condition 3. LZ77 tries to improve the match using lazy matching if and only if lazy matching is enabled and $|R| + 1 < \mathsf{max_lazy}$. Suppose lazy matching is enabled and $|R| + 1 < \mathsf{max_lazy}$, then by condition 1, $k = |T| - 1$, and by condition 4, LZ77 cannot find a match strictly longer than $S[1 : \ell_S - k]$ of length $\ell_S - k = |R| + 1$ at $S[1]$. Therefore, LZ77 cannot improve the match for $S[0 : |R|]$ using lazy matching. Therefore, LZ77 outputs a back-reference for $S[0 : |R|]$ and moves the pointer to $S[|R| + 1] = T[1]$. We discuss two possible cases depending on whether LZ77 finds the target string $T = S[|R| + 1 : \ell_S]$ in the sliding window. Note that condition 5 implies that the substring $S[|R| + 2 : \ell_S + 1]$ does not appear in the sliding window, which in turn implies that the substrings $S[|R| + 1 : \ell_S + 1]$ and $S[|R| + 2 : \ell_S + 2]$ do not appear in the sliding window.
  - If LZ77 can find T in the sliding window, then LZ77 finds a match for $S[|R| + 1 : \ell_S] = T$, which by condition 5 cannot be extended or improved by lazy matching. Therefore, LZ77 outputs a back-reference for $S[|R| + 1 : \ell_S]$ and moves the pointer to $S[\ell_S + 1]$, corresponding to the state true.
  - Otherwise, if LZ77 cannot find T in the sliding window, then by condition 1, LZ77 finds a match $S[|R| + 1 : \ell_S - 1] = T[1 : |T| - 1]$ in the sliding window, which cannot be extended. By condition 5, LZ77 also cannot improve the match using lazy matching. Therefore, LZ77 outputs a back-reference for $S[|R| + 1 : \ell_S - 1]$ and moves the pointer to $S[\ell_S]$, corresponding to the state false.

To summarise, the OR-match gadget presented above correctly updates the state of the automaton according to the disjunction of its initial state and the result of a match gadget on target string T.

Alternative construction. An analogy to the NOT gadget suggests that there should be alternative constructions of an OR-match gadget for deflate_fast and deflate_slow respectively. Here we only describe a construction based on lazy matching, and we omit the formal requirements and the analysis. Compared to our main construction, this construction allows k to be much smaller than $|T|$ when max_lazy is large, giving the attacker more freedom in adjusting the parameter k. This is useful if the requirements of our main construction cannot be satisfied when $k = |T| - 1$. However, this construction also has other more stringent requirements (in particular, $|T| \geq \mathsf{min_match} + 2$) and is generally more error-prone.

Assume the compressor uses the deflate_slow compression strategy. Let T be the target string, where $\mathsf{min_match} + 2 \leq |T| \leq \mathsf{max_match}$. Let k be an integer chosen by the attacker such that

$$\mathsf{min_match} \leq k \leq |T| - 2.$$

The OR-match gadget S we construct for the target string T is a byte string $S = R \| T$ of length $\ell_S = |R| + |T|$. Here R is a byte string chosen by the attacker such that

$$\mathsf{min_match} \leq |R| + 1 \leq \mathsf{max_lazy}$$

and

$$|R| + |T| - k - 1 \leq \mathsf{max_match}.$$

In addition, we require that if $|R| = 2$, then $|R| + |T| - k - 1 \geq \min\{\mathsf{max_lazy}, |T| - 1\}$.

We add three byte strings

$$\begin{aligned} S[0 : |R|] &= S[0] \| R \\ S[2 : \ell_S - k] &= R[2 : |R|] \| T[1 : |T| - k] \\ S[\ell_S - k + 1 : \ell_S] &= T[|T| - k + 1 : |T|] \end{aligned}$$

of length $|R|+1$, $|R|+|T|-k-1$ and k respectively to one of the two dictionaries.[128] We also add $S[|R|+1:\ell_S-1]=T[1:|T|-1]$ to one of the dictionaries, unless LZ77 can already find $T[1:|T|-1]$ in the sliding window. Because deflate_slow inserts every substring position in the sliding window into the LZ77 hash table, the order of these strings in the dictionaries does not matter.

One can easily check that setting $k=|R|=$ min_match meets our requirements on the parameters, but better parameter choices generally exist.

In our example setting, after the match gadget `secret=1`, an OR-match gadget for the target string `secret=2` can be `HUOsecret=2`, and we add three byte strings `1HUO`, `UOse` and `cret=2` into dict_2; here the attacker chooses $k=6$ and $R=$ `HUO`. The query (omitting the amplify gadget) can be

```
-1HUO-UOse-cret=2-secret=1HUOsecret=2
```

where we use `-` as the separator in dict_2 and omit dict_1 and filler.

We discuss the likely behaviours of LZ77:

- If the initial state of the OR-match gadget is true, which is expected to happen if $ck[1]=$ `1`, then LZ77 likely outputs `H<L4,D*><L6,D*>` for `HUOsecret=2` and moves the pointer to the byte following the last byte `2`, corresponding to the state true.
  We elaborate on the likely behaviour of LZ77. At the initial pointer position `H`, LZ77 finds a match for `HUO` in the sliding window, which LZ77 subsequently discards in favour of the longer match `UOse` found using lazy matching, so LZ77 outputs `H<L4,D*>` for `HUOse` and moves the pointer to `c`. Finally, LZ77 outputs a back-reference `<L6,D*>` for `cret=2` and moves the pointer to the byte following `2`.
- If the initial state of the OR-match gadget is false, which is expected to happen if $ck[1]\neq$ `1`, then the behaviour of LZ77 depends on the presence of the target string in the sliding window, which in turn depends on whether $ck[1]=$ `2`:
  - If LZ77 can find the target string `secret=2` in the sliding window, then LZ77 likely finds matches `1HUO` and `secret=2`, outputs `<L4,D*><L8,D*>` for `1HUOsecret=`, and moves the pointer to the byte following `2`, corresponding to the state true.
  - Otherwise, LZ77 likely finds matches `1HUO` and `secret=`, outputs `<L4,D*><L7,D*>` for `1HUOsecret=`, and moves the pointer to `2`, corresponding to the state false.

8.6 Example automata

Having described different types of gadgets, we are now ready to present some example automata. We design three automata that can be used to enhance compression side-channel attacks, the *CRIME chain*, *slide* and *cascade*. These automata should be sufficient for general-purpose compression side-channel attacks, although automata tailored to individual attack settings may have better performance.

For simplicity, we only describe the composition of gadgets when we specify an automaton, because the dictionary entries are determined by the gadgets in the automaton. For each automaton we describe below, we require that the amplify gadgets in the automaton are all of the same type, i.e. either all forward-amplify gadgets or all reverse-amplify gadgets. Without loss of generality, we assume the amplify gadgets we use are all forward-amplify gadgets, i.e. they compress to shorter lengths if the initial state is true. The match gadgets can also be replaced with unmatch gadgets or telescoped-match gadgets.

Note that an automaton may not perform its intended functionality if any requirement on its constituent gadgets is not satisfied. The attacker needs to perform checks on the constructed automata to avoid interference. For example, if an automaton contains two match gadgets for strings T_1 and T_2, with T_1 being a strict substring of T_2, then the match gadget for T_2 generally should not precede the match gadget for T_1. Ultimately, the validity and performance of an automaton need to be verified through experiments: we report on such experiments in Section 9.

CRIME chain. A CRIME chain allows the attacker to check with high confidence whether a target string T appears in the sliding window. For example, the attacker can check whether $ck[1]=g$ with the target string $T=P\|g$. We have already used CRIME chains implicitly in Sections 6 and 7.

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

$$\mathsf{align}\|\mathsf{match}\|\mathsf{amplify},$$

[128] Note that $|R|+|T|-k-1\geq|R|+1$.

where align, match and amplify are respectively an optional align gadget, a match gadget and an amplify gadget. Recall that the attacker cannot directly observe the state of the automaton, so we append an amplify gadget after the match gadget to make the match result visible from the output. To this end, the amplify gadget should create a compressed length difference that outweighs the noise in the output.

If LZ77 can find the target string T in the sliding window, then the amplify gadget has an initial state of true and decreases the compressed length; otherwise, the amplify gadget has an initial state of false and increases the compressed length. Therefore, the output, i.e. the response from the compressed length oracle, is expected to be shorter if LZ77 can find the target string T in the sliding window.

CRIME slide. A CRIME slide can be seen as a variant of the divide-and-conquer technique for compression side-channel attacks (Section 4.2) with amplified compressed length difference. It allows an attacker to check with high confidence whether at least one of the n target strings $T_1, \ldots, T_n$ appears in the sliding window. For example, using a CRIME slide, the attacker can perform a parallel search on multiple guesses $g_1, \ldots, g_n \in \Sigma$ for the secret byte $ck[1]$, with $T_i = P \| g_i$ for each $i \in [n]$.

The gadgets of a CRIME slide are

$$\mathsf{align} \| \mathsf{match}_1 \| \mathsf{OR\text{-}match}_2 \| \mathsf{OR\text{-}match}_3 \| \cdots \| \mathsf{OR\text{-}match}_n \| \mathsf{amplify},$$

where align is an optional align gadget, match_1 is a match gadget for the first target string T_1, $\mathsf{OR\text{-}match}_i$ is an OR-match gadget for the i-th target string T_i for $i \in \{2, \ldots, n\}$, and amplify is an amplify gadget.

Suppose LZ77 can find a target string T_k in the sliding window for some $k \in [n]$, and, without loss of generality, suppose LZ77 cannot find $T_1, \ldots, T_{k-1}$ in the sliding window. It is easy to see that the initial state of match_1 is true, the initial states of $\mathsf{OR\text{-}match}_2, \ldots, \mathsf{OR\text{-}match}_k$ are false, the initial states of $\mathsf{OR\text{-}match}_{k+1}, \ldots, \mathsf{OR\text{-}match}_n$ are true, and the initial state of amplify is true. As a result, the amplify gadget decreases the compressed length, likely yielding a short output. Otherwise, if LZ77 cannot find any target string in the sliding window, then the initial state of match_1 is true, the initial states of $\mathsf{OR\text{-}match}_2, \ldots, \mathsf{OR\text{-}match}_n$ are false, and the initial state of amplify is false. As a result, the amplify gadget increases the compressed length, likely yielding a longer output.

Note that the automaton excluding the amplify gadget should not be longer than the sliding window, in order for the gadget $\mathsf{OR\text{-}match}_n$ to work correctly. This is rarely a problem in practice, as OR-match gadgets are generally very short. We also note that the OR-match gadgets may influence the output by a small amount, especially if n is large.

We reiterate the possibility of interference among gadgets with the following example. Recall that, at each pointer position, LZ77 checks at most max_chain entries of a hash chain to find a match. If $n >$ max_chain, then a CRIME slide with $T_i = P \| g_i$ for each $i \in [n]$ might not work as intended. The reason is that the match gadget and earlier OR-match gadgets likely fill the hash chain for the hash value $P[1 : \mathsf{min_match}]$, such that LZ77 cannot find the position of $P \| ck$ in the plaintext in the hash table when its pointer is at subsequent positions of gadgets.

CRIME cascade. A CRIME cascade not only detects the presence of multiple target strings in parallel, but also allows the attacker to identify the first matched target string from the output. More specifically, a CRIME cascade for n target strings $T_1, \ldots, T_n$ allows the attacker to check with high confidence whether at least one target string appears in the sliding window, and if so, the attacker also learns the target string T_k with the smallest index that appears in the sliding window. For example, using a CRIME cascade with $T_i = P \| g_i$ for each $i \in [n]$, the attacker can check which of the guesses $g_1, \ldots, g_n \in \Sigma$ are correct for the secret byte $ck[1]$ or detect that none are correct.

The gadgets of a CRIME cascade are

$$\mathsf{align} \| \mathsf{match}_1 \| \mathsf{amplify}_1 \| \mathsf{OR\text{-}match}_2 \| \mathsf{amplify}_2 \| \cdots \| \mathsf{OR\text{-}match}_n \| \mathsf{amplify}_n,$$

where align is an optional align gadget, match_1 is a match gadget for the first target string T_1, $\mathsf{OR\text{-}match}_i$ is an OR-match gadget for the i-th target string T_i for $i \in \{2, \ldots, n\}$, and $\mathsf{amplify}_i$ is an amplify gadget for $i \in [n]$.

To see why this works, assume for simplicity that each amplify gadget decreases the compressed length by d bytes if the initial state is true, compared to the case where its initial state is false. If no target string appears in the sliding window, then the initial states of all amplify gadgets are false. Otherwise, if for some $i \in [n]$, the i-th target string T_i is the one with the smallest index in the sliding window, then for each $j \in [n]$, the initial state of $\mathsf{amplify}_j$ is true if and only if $j \geq i$, meaning that the compressed length is decreased by around $(n - i + 1) \cdot d$. Therefore, the attacker can learn from the output how many amplify

gadgets have an initial state of true, from which they can deduce whether any target string appears in the sliding window, and if so, the first matched target string.

Similar to a CRIME slide, the automaton excluding the final amplify gadget $\mathsf{amplify}_n$ should not be longer than the sliding window, in order for the gadget $\mathsf{OR\text{-}match}_n$ to work correctly. It is not required that each amplify gadget creates the same compressed length difference. The CRIME cascade works correctly as long as the compressed length difference of each amplify gadget outweighs the noise in the output, such that the attacker can uniquely identify the first matched target string from the output.

Alternative constructions. Even without using OR-match gadgets,[129] the attacker can still imitate a CRIME slide or a CRIME cascade by composing multiple CRIME chains, by either concatenating the gadgets or directly concatenating the automata. However, these constructions require significantly longer queries and are more prone to errors.

An attacker can construct the gadgets as

$$\mathsf{align}_1\|\mathsf{match}_1\|\mathsf{amplify}_1\|\mathsf{align}_2\|\mathsf{match}_2\|\mathsf{amplify}_2\|\cdots\|\mathsf{align}_n\|\mathsf{match}_n\|\mathsf{amplify}_n,$$

where for each $i \in [n]$, align_i is an align gadget, match_i is a match gadget for the i-th target string T_i, and match_i is an amplify gadget. Note that the align gadgets are not optional. For each $i \in [n]$, the initial state of match_i is true if and only if LZ77 can find the target string T_i in the sliding window. If we assume that every amplify gadget decreases the compressed length by around the same amount if the initial state is true, then the attacker can deduce from the output how many target strings are present in the sliding window, imitating a CRIME slide.[130] Furthermore, if each amplify gadget changes the compressed length by a different amount, then the attacker can learn more information from the output, or even achieve the functionality of a CRIME cascade and deduce from the output which target string is in the plaintext using a single query.

It is clear that this construction produces a longer query than a CRIME slide of similar effects, because it requires n amplify gadgets instead of one. Next, we informally argue that this construction produces a longer query than a CRIME cascade of similar effects. The intuition is that, while both constructions use n amplify gadgets, the amplify gadgets in a CRIME cascade can be made much shorter, since a single match may "activate" multiple amplify gadgets. We make a simplifying assumption that the compressed length difference arising from an amplify gadget is linear in its length; that is, there exists a positive number λ, such that any amplify gadget that creates a compressed length difference of ℓ bytes is of around $\lambda\ell$ bytes for any reasonable ℓ. To achieve a difference of at least d bytes between any two outputs that are intended to be different, the total length of amplify gadgets should be at least roughly λdn bytes in a CRIME cascade, and it should be at least about $\lambda dn(n+1)/2$ bytes in this construction. Note that, even if the attacker is allowed to make arbitrary long queries, the query (excluding $\mathsf{amplify}_n$) still needs to be shorter than the sliding window size $\mathsf{window_size}$ in order for match_n to work correctly.

The attacker can also directly concatenate multiple CRIME chains by making a query

$$Q = \mathsf{crime\text{-}chain}_1\|\mathsf{crime\text{-}chain}_2\|\cdots\|\mathsf{crime\text{-}chain}_n,$$

where for each $i \in [n]$, $\mathsf{crime\text{-}chain}_i$ represents the CRIME chain constructed for the i-th target string T_i. This construction achieves a similar effect as the construction above and shares similar limitations.

9 Experimental evaluation

9.1 Implementation

We implemented a framework for performing compression side-channel attacks using CRIME automaton in Python 3, which also incorporates our amplification techniques in Sections 6 and 7. Our framework is open source at https://github.com/yymmsong/crime-automata.

In Sections 6 to 8, we described many different techniques for compression side-channel attacks, and some of these techniques are highly complex (e.g. collision-based amplification). Consequently, our

[129] It could be that OR-match gadgets cannot be used for some reason, e.g. LZ77 cannot find $T[|T|-k+1:|T|]$ in the sliding window for the target string T because of hash table collisions or other types of interference. More generally, it is easier to adapt an automaton without logical gadgets to other compression algorithms.

[130] In fact, this construction is more powerful than a CRIME slide, which does not allow the attacker to learn the number of matched target strings.

framework has several interesting features that lower the barrier to evaluate and apply our techniques, which we outline below. However, we stress that our framework cannot be directly used to attack real-world systems: any such attempt requires the implementation of a compression length oracle as an interface between the target system and our framework, which is often non-trivial, and it may also be necessary to adapt our techniques to the real-world attack scenario.

Modularity and compilation. The CRIME automaton paradigm naturally allows modularity in our framework. Our implementation of an automaton maintains a list of gadgets, and the user can insert, remove or replace gadgets at will. For example, the attacker may want to change their guess for the secret byte in a CRIME chain while minimally affecting other parts of the query, and this can be done in our framework by replacing the match gadget corresponding to the guess.

An implementation issue is that a gadget may add a byte string that contains the last byte of its previous gadget to a dictionary (e.g. all amplify gadgets), so adding, removing or replacing a gadget may also change the behaviour of its subsequent gadget. Our solution is to introduce the concept of compilation for CRIME automata. In our framework, every gadget specifies its content and a list of actions, such as adding a byte string that contains the unspecified preceding byte to a dictionary. On compilation, the automaton sequentially executes the actions of each gadget to construct dict_1 and dict_2, concatenates the content of each gadget to obtain $\mathsf{gadgets}$, and constructs filler as specified. The compiled automaton is the concatenation of dict_1, filler, dict_2 and $\mathsf{gadgets}$.

Many gadgets are partially or entirely constructed using random bytes, and a desirable functionality is to instantiate a gadget with different randomness, for example when the gadget does not work correctly. Similarly, it is useful to change some parameters of a gadget in place. We allow greater freedom to design and adjust CRIME automata in our framework by making the gadgets compilable, thereby separating the process of defining and instantiating a gadget. On compilation, the unspecified parts of a gadget are selected automatically, such as random parts of the content and unspecified parameters, generating output and actions that may differ across compilations. Compiling an automaton triggers the compilation of its gadgets that have not been compiled, unless the compilation explicitly forces all gadgets to compile. One can also recompile some gadgets in an automaton while keeping others unchanged.

Automation. Our framework reduces the burden on users to understand our techniques and parse their often complex requirements by enabling a high level of automation.

A gadget often has several parameters, which are important for its correctness or performance. However, it can be laborious to understand the purpose and effect of each parameter and identify its requirements from our description. Therefore, our framework enables the automated selection of some parameters of a gadget on compilation. The automated parameter selection is based on a set of hard-coded rules, and it may depend on the DEFLATE compressor configuration that the user specified for the automaton and other parameters of the gadget. We remark, however, that adjusting the parameters manually can often yield better results.

We also implement a functionality that we refer to as sanity checking, which detects obvious errors in the specification of a gadget by checking whether its parameters violate some of the requirements. A by-product of sanity checking is the automated selection from different constructions of the same gadget that are intended for different compressor configurations (e.g. match-based chaining in Section 6.3.2). The user can initialise a gadget that simultaneously specifies a list of candidate gadgets, and the first candidate gadget that passes sanity checking will be instantiated on compilation.

Finally, our framework allows users to construct collision-based amplification chains directly from collisions. While we have devoted a large part of Section 7 to reducing the task of constructing collision-based amplification chains to finding suitable instance sets, it is still difficult for users to manually construct such instance sets in the first place without carefully studying Section 7. Therefore, we also implement a method for automatically constructing suitable instance sets from diverse colliding or double-colliding sets. Specifically, we first heuristically determine k, m and the length of each element for the instance set, and then repeat the process of constructing an instance set with the specified layout by appending random bytes to the collisions, until the requirements on the instance set are satisfied or a preset number of iterations have been reached. We note that this method may not always work correctly and generally results in instance sets that perform worse than those carefully constructed manually.

Validity check. As readers have seen in Sections 6 to 8, some of our techniques have extensive prerequisites, which pose challenges to implementation and application alike. In addition, our requirements are often harder to ensure than they appear; for example, whether LZ77 can find a given byte string in

the sliding window depends on the hash table implementation and the compressor configuration, among other things. On the other hand, many requirements are largely theoretical and only in place to enable formal analyses, and either they are satisfied with high probability or they can be significantly weakened or dropped without affecting the correctness of the gadget. Therefore, it is more fitting to interpret these requirements as demonstrating that the construction likely works under suitable parameters than strictly enforcing them.

Our alternative to enforcing the theoretical requirements is to test the constructed query in a simulated attack environment and see if they produce desired results, which we refer to as a validity check. That is, the user can create test oracles that, to the best of their knowledge, behaves like the real-world compression length oracle that they target, and see if the responses to the query from the test oracles are similar to what they expect. Our framework provides a template test oracle that the user can instantiate by setting appropriate parameters.

To perform a validity check, the user may submit the same query to different test oracles constructed with different secrets, layouts or noise. For example, the expected behaviour of a CRIME chain query is that there is a significant difference between the responses from a test oracle that embeds the target string in the plaintext and a similar test oracle that does not. If the validity check succeeds, then the attacker submits the query to the real-world oracle, which is likely to give desired responses if the simulation is accurate enough; otherwise, the attacker generates another query with different randomness and repeats the above process, until a preset number of attempts has been reached. We implemented validity checks for the example automata in Section 8.6.[131]

Attack strategies. We focus on two types of attack strategies for secret recovery in our framework: the *batch searcher* and the *cascade searcher*. A batch searcher uses CRIME slides to perform divide-and-conquer attacks. A cascade searcher, on the other hand, uses generalised CRIME cascades to speed up secret recovery, and therefore performs fewer queries than batch searchers in general. More specifically, the cascade searcher may place more than one match and OR-match gadget before each amplify gadget in a CRIME cascade, so that the oracle response reflects which subset of targets contains the correct guess. This helps the attacker balance the number of queries and query lengths.[132] Note that fine-tuning the attack strategies for compression side-channel attacks against specific targets is not the focus of this paper.

9.2 Experimental setup

We evaluate our techniques in a simulated setting conforming to our attack model in Section 5.1. Unless otherwise specified, we use the following setup. Each experiment is repeated for 1000 independent rounds. In each round, the attacker has access to a compression length oracle

$$|\mathsf{C}(\mathsf{data}_1 \| P \| ck \| \mathsf{noise} \| \mathsf{data}_2 \| Q \| \mathsf{data}_3)|,$$

where C is the zlib compressor with default configuration; data_1, data_2, data_3 are ASCII-printable byte strings of 500 bytes each, unknown to the attacker; P is an 8-byte alphanumeric prefix; ck is an alphanumeric secret of later-specified length; noise is a random byte string whose length ℓ_{noise} follows a uniform distribution on $\{0, \ldots, L_{\mathsf{noise}}\}$ for a later-specified noise level, with ℓ_{noise} and noise re-sampled for each oracle query; and Q is an ASCII-printable query chosen by the attacker.[133]

At the start of each round, data_1, data_2, data_3, P and ck are sampled uniformly at random, and the attacker is given P. For experiments on amplification, $|ck| = 1$, and we measure the difference in oracle responses to queries for a correct guess and a random incorrect guess. For experiments on attacks, $|ck| = 32$. The attacker returns their guess ck' for the secret, and they succeed if and only if $ck = ck'$.

Note that our model as specified does not capture noise added after compression. Intuitively, noise within the plaintext has a greater effect over the oracle responses, because it alters the behaviour of the compressor and generally expands in length because of Huffman coding. We report on the impact of block cipher padding in Section 9.3.

[131] For evaluation, we disallow the attacker from performing validity checks on the generated queries, except for determining certain parameters for our attack strategy using CRIME cascades prior to the actual attack.

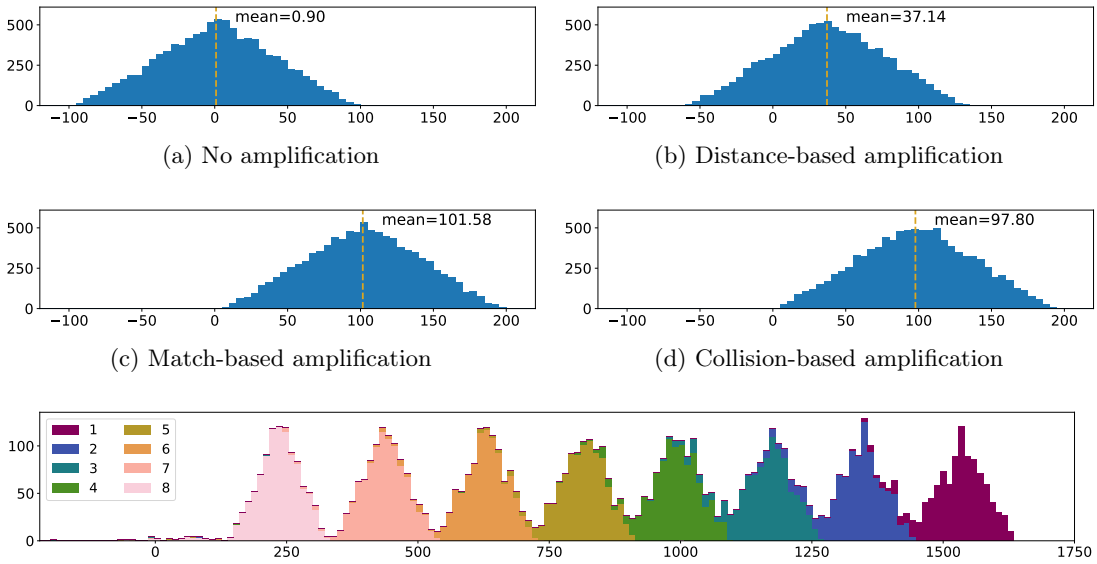
[132] We also implement an optimisation that we refer to as *mirroring*: in order to reduce the dictionary sizes, we instantiate all collision-based amplify gadgets in one query in essentially the same way, so that each amplify gadget acts as a dictionary for its subsequent amplify gadget; see our implementation for more details.

[133] More accurately, we configured our framework such that the queries only contain alphanumeric bytes and two symbols "`-`" and "`@`".

9.3 Results

9.3.1 Amplification

We first run experiments on our amplification techniques. In each round of an experiment, we generate two CRIME chain queries (in random order), one for the correct guess and another for a random incorrect guess. Each round uses fresh randomness to construct the amplify gadget. We report the difference in compression length oracle responses between the incorrect query and the correct query.[134] Figures 10a to 10d demonstrate the effectiveness of amplification with 0-50B uniform noise. Using queries of at most 1KiB in length, distance-based amplification already partially offsets the noise (Fig. 10b), while match-based and collision-based amplification almost completely outweigh it (Figs. 10c and 10d).



(a) No amplification

(b) Distance-based amplification

(c) Match-based amplification

(d) Collision-based amplification

(e) CRIME cascade with collision-based amplification

Fig. 10: Histograms of the compressed length difference (x-axis, in bytes) for different amplification techniques, with 0-50B uniform noise. In (a)-(d), the maximum query length L_Q is 1KiB, and we perform each experiment for 10000 rounds. In (a), the query is simply $P\|g$ for some guess g. Experiments (b)-(d) use CRIME chains. In (b), we set $k = 4$ and use doubling. In (c), we set $k = 5$ and use doubling. In (d), we truncate the collision-based amplify gadget to meet the length limit. In (e), we evaluate CRIME cascades with 8 collision-based amplify gadgets with 0-50B uniform noise. For each $i \in [8]$, label i represents the compressed length difference depending on whether the i-th target is correct, the other 7 targets being random incorrect guesses. The maximum query length in (e) is 9221 bytes. Note that (e) omits one occurrence of an outlier.

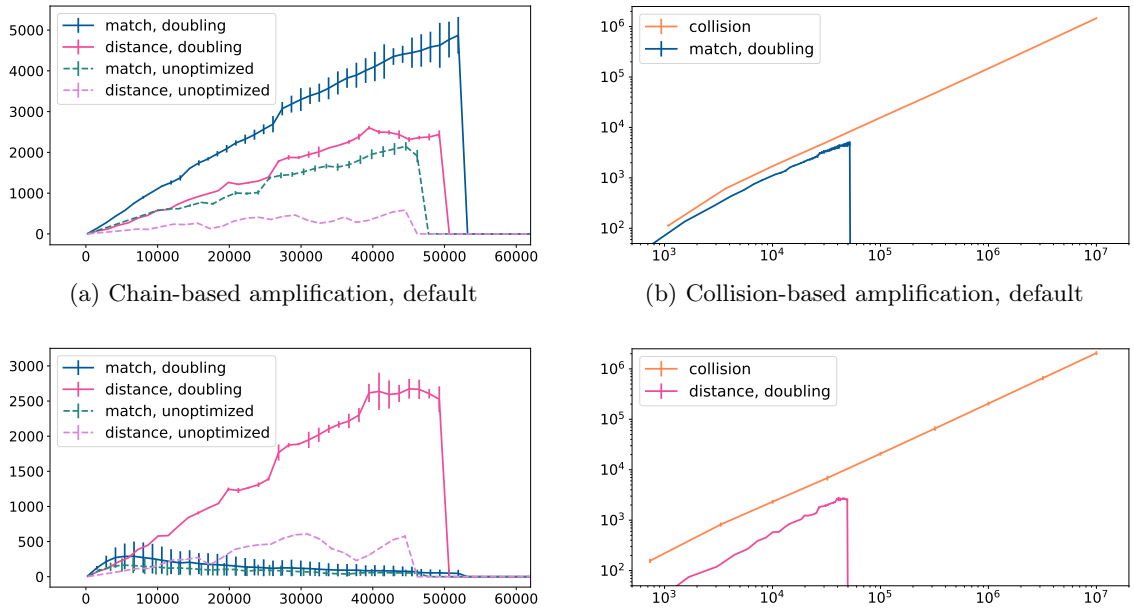
To demonstrate the effectiveness of amplification in CRIME cascades, the experimental setup is similar: in each round, we generate a new CRIME cascade that tests 8 targets simultaneously using collision-based amplification. The correct query is obtained by setting $P\|ck$ as the i-th target, and we repeat the experiment for each $i \in [8]$. Figure 10e shows that, even with 0-50B noise, one can infer from the compressed length difference with high probability whether any target is matched and, if so, which one corresponds to the correct guess.

Query length. In Figs. 11a and 11b, we show how the compressed length changes with the query length for different amplification techniques. While all of our techniques can create large compressed length differences, they do so at different rates, with collision-based amplification being the most efficient in general. In Fig. 11a, the compressed length difference vanishes when the query length reaches around 50KiB, because the dictionaries become too long such that the match gadget cannot find $P\|ck$ in the

[134] For clarity, we reverse the directions of compressed length differences when evaluating reverse-amplify gadgets.

sliding window. This phenomenon does not occur in collision-based amplification (Fig. 11b), whose dictionary size does not grow with the query length.

In Figs. 11c and 11d, we show the effect of the query length for different amplification techniques in zlib compression level 3, which uses the deflate_fast compression strategy. While the performance of distance-based and collision-based amplification is similar to zlib default, match-based amplification is only effective for short queries (Fig. 11c), because its correctness relies on the absence of certain 3-byte strings in the sliding window; see Section 6.3.2.



(a) Chain-based amplification, default

(b) Collision-based amplification, default

(c) Chain-based amplification, level 3

(d) Collision-based amplification, level 3

Fig. 11: The average compressed length difference (y-axis) with respect to the query length ℓ_Q (x-axis) for different amplification techniques. An error bar represents plus/minus one sample standard deviation. We use the compression level 6 (default) in (a)-(b) and level 3 in (c)-(d). We use $k = 5$ for distance chains and $k = 6$ for match chains. For better comparison, we plot the evaluation results for match chains (resp. distance chains) using doubling in both (a) and (b) (resp. (c) and (d)).

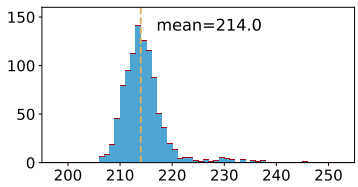
Other DEFLATE implementations. To demonstrate the generalisability of our amplification techniques, we focus on collision-based amplification, which depends on the hash function used in the DEFLATE implementation. In addition to zlib, we examined GNU Gzip, zlib-ng, the Cloudflare fork of zlib, the Chromium fork of zlib, zlib in Go, and flate2 in Rust. Except for zlib, GNU Gzip and flate2 in Rust, each compressor uses a different non-rolling hash function. Our framework can automatically construct amplify gadgets from a few suitable collisions. Assuming default configurations, the performance of collision-based amplification under these compressors is similar to that under zlib, except for flate2 in Rust: for example, for queries of about 10^5 bytes, the average compressed length differences are respectively 1.57×10^4 B, 1.74×10^4 B, 1.56×10^4 B, 1.44×10^4 B and 1.44×10^4 B under GNU Gzip, zlib-ng, the Cloudflare fork of zlib, the Chromium fork of zlib and zlib in Go.

We now briefly discuss the performance of collision-based amplification under flate2 in Rust. Recall from Section 3.5 that the default backend of flate2 in Rust is miniz_oxide, a Rust port of miniz, and most collision-based amplification constructions cannot achieve unlimited amplification against miniz. Indeed, with our main template in Section 7.3.2, the average compressed length differences are 1.03×10^4 B for queries of around 10^5 bytes and 5.47×10^3 B for queries of around 10^7 bytes. Nevertheless, by applying the alternative instance set constraints in Section 7.3.6, the performance of collision-based amplification under flate2 in Rust becomes comparable to that under zlib, albeit still less efficient; in particular, the

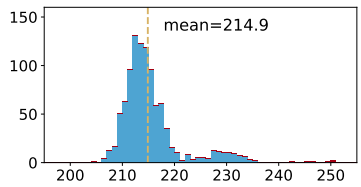
average compressed length differences are 9.93×10^3 B for queries of around 10^5 bytes and 8.00×10^5 B for queries of around 10^7 bytes.[135]

9.3.2 Secret recovery

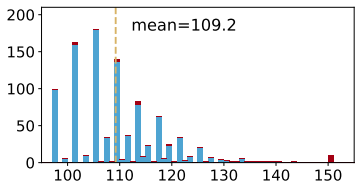
Figures 12a to 12d demonstrate the effectiveness of CRIME automata in secret recovery attacks. Even with up to 100 bytes of noise, the batch searchers can almost always recover the secret, and the number of queries performed is close to that of an ideal binary search algorithm (Figs. 12a and 12b). The cascade searchers almost halve the number of queries made by binary searchers, while still recovering the secret most of the time (Figs. 12c and 12d).



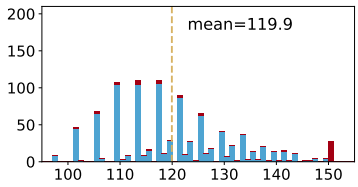
(a) Batch match searcher, 0-50B noise. No failures, max. query length 2936B.



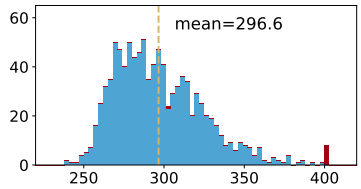
(b) Batch collision searcher, 0-100B noise. 3 failures, max. query length 3749B.



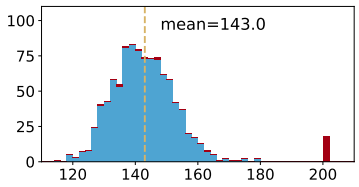
(c) Cascade match searcher, 0-50B noise. 37 failures, max. query length 9896B.



(d) Cascade collision searcher, 0-100B noise. 66 failures, max. query length 17265B.



(e) Level 3 batch match searcher, 0-50B noise. 9 failures, max. query length 2415B.



(f) Level 3 cascade collision searcher, 0-100B noise. 27 failures, max. query length 21372B.

Fig. 12: Histograms of the number of queries performed (x-axis) for recovering a 32-byte uniform alphanumeric byte secret. The blue bars represent successful experiments, while the red bars represent failed experiments. We use "match" and "collision" to denote match-based and collision-based amplification. We empirically set the maximum number of queries to resp. 250, 250, 150, 150, 400, 200 in (a)-(f). We use the compression level 6 (default) in (a)-(d) and level 3 in (e)-(f). For comparison, binary search without noise in responses requires about $\lceil 32 \cdot \log_2 62 \rceil = 191$ queries to recover a secret.

Attacking deflate_fast. We also evaluate our attacks on zlib compression level 3, which uses the deflate_fast compression strategy. A challenge is that max_chain = 32 on level 3, so we cannot test more than 16 guesses in a single query without meticulously configuring the match and OR-match gadgets. Nevertheless, even limiting the maximum number of targets per query to 15, our attacks still work well on zlib level 3. As shown in Figs. 12e and 12f, the success rates and maximum query lengths are similar to zlib default (Figs. 12a and 12d), with only a moderate increase in the number of queries.

Impact of block cipher padding. In practice, the compressed data is often padded to block cipher boundaries before encryption, causing the compression length oracle to return inexact lengths. Previous works addressed this by adding fillers to the queries to align the compressed data to block boundaries, which requires a careful selection of fillers and significantly increases the number of queries [GHP13, KZ16]. In comparison, our techniques can easily outweigh the inaccuracies in oracle responses with a large

[135] We set $\ell = 12$ in our construction. We stress that we assume here knowledge of the query position in the plaintext, which holds in our experimental setup when no noise is present. Otherwise, as discussed in Section 7.3.6, one would need to randomise the starting position of the query and potentially use a larger ℓ, both leading to worse performance.

enough amplification. We also performed the attack experiments above with the additional restriction that the oracle responses are always rounded up to the next multiple of 16 bytes. The results are very similar to Figs. 12a to 12f, with the average number of queries being 213.8, 215.0, 109.5, 121.3, 296.1 and 142.5 respectively.

Comparison with Rupture. We extracted the relevant backend code from the Rupture framework [KZ16] and adapted it to our attack model. Because Rupture cannot recover the secret in the settings of Figures 12a to 12d within a reasonable number of queries, we ran experiments in a considerably less challenging scenario, where the attacker tries to recover a one-byte hexadecimal secret with 0-10B uniform noise.[136] As shown in Section 9.3.2, both of our attackers significantly outperform Rupture both in terms of success rate and query count, though they use longer queries. Notably, a cascade searcher recovers the secret with *two* queries on average, whereas Rupture makes about 700 times as many queries.

Method	Success rate	Mean number of queries	Maximum query length
Rupture	222/1000	1377.0	97B
Batch	991/1000	5.6	615B
Cascade	962/1000	2.0	3624B

Table 5: Comparison of Rupture and our techniques on recovering one hexadecimal byte with 0-10B uniform noise.

10 Discussion

As we have demonstrated, our techniques could potentially be used to improve most existing compression side-channel attacks against DEFLATE, and they are likely to be especially effective against systems that are noisy or not highly interactive (e.g. [BME25, FNA+24, FPN+24]). This means that systems performing compression before encryption may pose greater risks than expected, and we urge vendors to mitigate the compression side channels in their systems. However, we note that real-world compression side channels do not always satisfy the prerequisites of our techniques, and the attacker may need to adapt our techniques due to possible deviations from our attack model.

Mitigations. While previous works have suggested that random-length padding can be effective at mitigating compression side-channel attacks [Deg21, PFSG22], and some vendors introduced noise to the compressor input [FNA+24] or output [Djab] as countermeasures, our techniques imply that these countermeasures are not as effective as may have been believed. The number of queries required for compression side-channel attacks when noise is present had been significantly overestimated [Deg21, GHP13,KZ16,PFSG22]. In particular, our collision-based amplification technique in Section 7 enables the attacker to offset any realistic amount of noise if there is no limit on the query length. Similarly, limiting interaction is also less effective against compression side-channel attacks than previously expected.

The most effective countermeasure against compression side-channel attacks is to disable compression. We are unaware of any other general-purpose mitigation that can prevent these attacks. It is therefore up to system designers to select the mitigations that best fit their systems and threat models. We refer readers to Section 2.3 for our survey of mitigations for compression side-channel attacks, including separating secrets, compartmentalization, switching compressor, masking secrets, adding noise, limiting interaction and combining compression with encryption. Other mitigations for compression side-channel attacks also exist, especially for individual systems such as web servers. Similar to other side channels, practical defences should target concrete compressor implementations, which usually behave differently from theoretical compression algorithms.

[136] We limit the maximum number of queries to 3000. For Rupture, we use the divide-and-conquer strategy and empirically set `samplesize` to 100 and `confidence_threshold` to 0.4 to strike a balance between the success rate and the number of queries.

It might be tempting for some vendors to deploy ad hoc mitigations for our attacks. Since our techniques require the precise knowledge of the DEFLATE compressor, any non-trivial modification to the compressor may invalidate some or even all of our techniques, such as not inserting each substring position to the hash table with a certain probability. However, we discourage solely relying on mitigations that have not received detailed analyses, as they may be ineffective for variants of our attacks, or worse, create new attack vectors.

Our techniques imply that the maximum query length is an important capacity of the attacker: longer queries enable greater amplification, and the attacker can reduce the number of queries by combining multiple queries into a single one. Therefore, one should take into account the maximum length of the data that the attacker can control when evaluating mitigations. However, it appears difficult to mitigate compression side-channel attacks by limiting the maximum query length alone, because restricting the potentially attacker-controlled fields would likely cause disruptions to normal users as well.

Other LZ77-based compressors. Many popular compression algorithms follow the general structure of DEFLATE, using variants of LZ77 in combination with entropy encoding algorithms like Huffman coding. Prominent examples include brotli [AFI23] and Zstandard [Met], which have seen increasing adoption in recent years and are supported in many different protocols and applications, e.g. HTTP compression [IAN]. In addition, some fast compression algorithms such as LZ4 [LZ4] and Snappy [Sna] only employ variants of LZ77 and do not perform Huffman coding, trading compression ratio for speed.

Since existing techniques for compression side-channel attacks mainly exploit LZ77 in DEFLATE, it is not surprising that compression side channels in other LZ77-based compressors exist and can be exploited in similar ways. Indeed, the VORACLE attack also targets LZ4 [Naf18], DBREACH attacks also work for LZ4 and Snappy, for which the attacker computes compressibility scores using different heuristics from zlib [HME23], and an informal analysis shows that brotli is vulnerable to a variant of Kelsey's adaptive input attack (Section 4.1).[137] The compressed format specifications of brotli [AS16] and Zstandard [CK21] both acknowledge possible compression side channels.

Similarly, because most of our techniques for compression side-channel attacks against DEFLATE in this paper exploit LZ77, one would naturally expect that they can also be applied to other LZ77-based compressors with little or no modification. However, there are several additional obstacles to attacking these compressors. First, LZ77-based compressors often make significant changes to the LZ77 algorithm to improve its performance, sometimes to the extent that they bear little resemblance to LZ77 in DEFLATE; for example, LZ77 at the highest compression level of brotli does not have a hash table or scan the input linearly with a pointer [AFF+19], invalidating our techniques in Sections 7 and 8. Second, LZ77-based compressors often perform complex engineering optimisations that include many heuristics-based implementation tricks, which make it considerably harder to devise attacks that rely on subtle details of compressor implementations, such as collision-based amplification in Section 7. Finally, unlike DEFLATE, which has been studied extensively, many LZ77-based compressors do not describe or document their compression algorithms in detail, so one has to rely on manual inspection of the source code to discover attacks.

Sensitivity. As we remarked in Section 7, our collision-based amplification technique shows that the additive sensitivity of LZ77 in DEFLATE is much higher than that of the theoretical LZ77 algorithm in [ZL77]. This means that determining the amount of random padding for DEFLATE based on the latter metric is insufficient to defend against our attacks. However, real-world compressors can be extremely complex, so it is very hard to find any compressor in common use with a low sensitivity. Conversely, designing a practical compressor with a low sensitivity likely rules out popular compression algorithms like LZ-family algorithms and the Burrows-Wheeler transform as well as many engineering optimisations, which likely lead to poor performance. We note that the study of compressor sensitivity is currently limited to deterministic compressors, and introducing non-determinism to the compression algorithm might be a way to build robust and efficient compressors.

Future work. There are several possible directions for future work. For instance, it remains an open problem to demonstrate the practicality of our techniques in proof-of-concept compression side-channel attacks against concrete systems, which likely require adaptations of our techniques to each individual attack scenario. Some of our techniques can likely be further refined, and it should be possible to find

[137] https://security.stackexchange.com/questions/172188/brotli-compression-algorithm-and-breach-attack

new techniques with a better understanding of the Huffman coding process in DEFLATE and implementations.

Further, it would be of particular interest to extend our results to other compressors. Some variants of our amplification techniques appear to apply to several other compressors, like brotli and bzip2. However, one may uncover better techniques by inspecting their algorithms and implementations more closely. Automated tools like fuzzing and machine learning may also help in that regard, following other side-channel attacks exploiting compression [SBS+23, MK24].

An interesting analogy to our work is to improve existing techniques for exploiting timing side channels in implementations of compression algorithms [SBS+23]. Sihlovec [Sih25] showed that our collision-based amplification technique can be used to amplify decompression timing differences in zlib, and proposed a technique for creating larger decompression timing differences by exploiting the block type assignment in zlib. Similarly, an in-depth examination will likely lead to more practical timing side-channel attacks against other compressor and decompressor implementations.

While many mitigations for compression side-channel attacks have been proposed, more work needs to be done in rigorously analysing and evaluating these mitigations. In addition, a research goal of practical significance is to lower the cost and engineering effort of implementing mitigations for compression side-channel attacks, thereby facilitating their adoption.

Despite previous works on formalising compression side-channel attacks [ASB15, FBNR25], including an earlier version of this paper [Son24], the current theoretical results still fail to answer many questions about the compression side channel, and more research is needed in this direction. For example, no security proof exists for the intuitively secure mitigation of masking secrets, and the effect of noise in the plaintext on compression side-channel attacks is unknown. At the same time, we call for more studies on the robustness of compression algorithms that are modelled more closely on real-world compressors. It is a challenging open problem to find or design a compressor that is both provably robust and practically efficient.

Finally, we concur with [Kel02] that the compression side channel is different in nature from many other side channels. Like encryption algorithms, compression algorithms are mathematical objects that can be understood and studied precisely. This enables a wider range of approaches than those commonly used in side-channel attacks. Furthermore, even in an idealised environment, no cryptosystem can achieve IND-CPA security and general-purpose compression at the same time [KT14]. We therefore advocate for a change of mindset when it comes to analysing compression side-channel attacks. In particular, a more fitting name for compression side-channel attacks could be *compression (length) oracle attacks*.

Acknowledgement

This work was partially supported by the Zurich Information Security and Privacy Center (ZISC).

References

ABJM21. Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Mesh messaging in large-scale protests: Breaking bridgefy. Cryptology ePrint Archive, Report 2021/214, 2021.

AEP22. 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.

AFF+19. Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. Brotli: A general-purpose data compressor. *ACM Trans. Inf. Syst.*, 37(1):4:1–4:30, 2019.

AFI23. Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *Inf. Comput.*, 291:104999, 2023.

AS16. Jyrki Alakuijala and Zoltan Szabadka. Brotli Compressed Data Format. RFC 7932, July 2016.

ASB15. 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, Berlin, Heidelberg, January 2015.

BAP+16. Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *SIGSOFT FSE*, pages 193–204. ACM, 2016.

BLG25. Jeremiah Blocki, Seunghoon Lee, and Brayan Sebastián Yepes Garcia. Differentially private compression and the sensitivity of LZ77. In Benny Applebaum and Huijia (Rachel) Lin, editors, *TCC 2025, Part IV*, volume 16271 of *LNCS*, pages 356–386. Springer, Cham, December 2025.

BLP+11. Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to "weird machines" and theory of computation. *login Usenix Mag.*, 36(6), 2011.

BME25. Britney Bourassa, Yan Michalevsky, and Saba Eskandarian. G-DBREACH attacks: Algorithmic techniques for faster and stronger compression side channels. In *ACNS (2)*, volume 15826 of *Lecture Notes in Computer Science*, pages 24–48. Springer, 2025.

Boy91. 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, Berlin, Heidelberg, April 1991.

BRB18. Lucas Bang, Nicolás Rosner, and Tevfik Bultan. Online synthesis of adaptive side-channel attacks based on noisy observations. In *2018 IEEE European Symposium on Security and Privacy*, pages 307–322. IEEE Computer Society Press, April 2018.

bro. brotli. https://web.archive.org/web/20260310170215/https://brotli.org/.

BS13. Tal Be'ery and Amichai Shulman. A perfect CRIME? only TIME will tell. *Black Hat Europe*, 2013.

BW94. Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. *SRS Research Report*, 124, 1994.

CBH93. Guy Castagnoli, Stefan Brauer, and Martin Herrmann. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. *IEEE Trans. Commun.*, 41(6):883–892, 1993.

chr. chromium. zlib. https://web.archive.org/web/20260309210526/https://chromium.googlesource.com/chromium/src/third_party/zlib/.

CK21. Yann Collet and Murray Kucherawy. Zstandard compression and the 'application/zstd' media type. RFC 8878, February 2021.

Clo. Cloudflare. Cloudflare fork of zlib. https://web.archive.org/web/20260221070105/https://github.com/cloudflare/zlib.

Deg21. 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.

Deu96a. L. Peter Deutsch. DEFLATE compressed data format specification version 1.3. RFC 1951, 1996.

Deu96b. L. Peter Deutsch. GZIP file format specification version 4.3. RFC 1952, 1996.

Djaa. Django. Cross Site Request Forgery protection. https://web.archive.org/web/20260227103259/https://docs.djangoproject.com/en/5.2/ref/csrf/.

Djab. Django. GZip middleware. https://web.archive.org/web/20260225072331/https://docs.djangoproject.com/en/dev/ref/middleware/#module-django.middleware.gzip.

DlG96. L. Peter Deutsch and Jean loup Gailly. ZLIB compressed data format specification version 3.3. RFC 1950, 1996.

EBE+21. Dmitry Evtyushkin, Thomas Benjamin, Jesse Elwell, Jeffrey A. Eitel, Angelo Sapello, and Abhrajit Ghosh. Computing with time: Microarchitectural weird machines. In *ASPLOS*, pages 758–772. ACM, 2021.

FBNR25. Andrés Fábrega, Samuel Breckenridge, Armin Namavari, and Thomas Ristenpart. Mitigating injection attacks against E2EE applications via view-based partitioning. In Lujo Bauer and Giancarlo Pellegrino, editors, *USENIX Security 2025*, pages 2965–2984. USENIX Association, August 2025.

Fel. Antaeus Feldspar. An explanation of the *Deflate* algorithm. https://web.archive.org/web/20260308000614/https://zlib.net/feldspar.html.

FLS23. 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, Cham, April 2023.

FNA+24. Andrés Fábrega, Armin Namavari, Rachit Agarwal, Ben Nassi, and Thomas Ristenpart. Exploiting leakage in password managers via injection attacks. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

FPN+24. Andrés Fábrega, Carolina Ortega Pérez, Armin Namavari, Ben Nassi, Rachit Agarwal, and Thomas Ristenpart. Injection attacks against end-to-end encrypted applications. In *2024 IEEE Symposium on Security and Privacy*, pages 2648–2665. IEEE Computer Society Press, May 2024.

Fre. Free Software Foundation, Inc. GNU Gzip. https://web.archive.org/web/20251219035408/https://www.gnu.org/software/gzip/.

Fry. Mike Frysinger. ncompress. https://web.archive.org/web/20260311165125/https://vapier.github.io/ncompress/.

FSNI25. Yuto Fujie, Hiroki Shibata, Yuto Nakashima, and Shunsuke Inenaga. Tight additive sensitivity on lz-style compressors and string attractors. In *SPIRE*, volume 16073 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2025.

GGK+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.

GHP13. Yoel Gluck, Neal Harris, and Angelo Prado. BREACH: Reviving the CRIME attack. *Black Hat*, 2013.

GIL+23. 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.

GJLN22. Kai Gellert, Tibor Jager, Lin Lyu, and Tom Neuschulten. On fingerprinting attacks and length-hiding encryption. In Steven D. Galbraith, editor, *CT-RSA 2022*, volume 13161 of *LNCS*, pages 345–369. Springer, Cham, March 2022.

GKO+25. Yoav Gross, Shmuel T Klein, Elina Opalinsky, Rivka Revivo, and Dana Shapira. Compression cryptosystem. *The Computer Journal*, 2025.

Gup17. Ankit Anand Gupta. Length hiding VPN to mitigate compression side-channel and traffic analysis attacks. Master's thesis, University of Minnesota, 2017.

HME23. 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.

HRY24. Gal Horowitz, Eyal Ronen, and Yuval Yarom. Spec-o-scope: Cache probing at cache speed. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 109–123. ACM Press, October 2024.

Huf52. David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

IAN. IANA. Hypertext Transfer Protocol (HTTP) parameters. https://web.archive.org/web/20260314172852/https://www.iana.org/assignments/http-parameters/http-parameters.xhtml.

JKS02. 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.

JWR04. 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, Berlin, Heidelberg, February 2004.

Kap23. David A. Kaplan. Optimization and amplification of cache side channel signals, 2023.

KB07. 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.

KBF22. Charles 'Buck' Krasic, Mike Bishop, and Alan Frindell. QPACK: Field compression for HTTP/3. RFC 9204, 2022.

Kel02. 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, Berlin, Heidelberg, February 2002.

KH25. Hyoungshick Kim and Sangwon Hyun. Prexion: Probabilistic matching-based compression to mitigate compression side channel attacks against HTTPS. *IEEE Access*, 13:86069–86082, 2025.

KHJ+10. 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.

KKC+23. Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: Improving cache attacks with transient execution. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1955–1972. USENIX Association, August 2023.

KKSZ16. Dimitris Karakostas, Aggelos Kiayias, Eva Sarafianou, and Dionysis Zindros. CTX: Eliminating BREACH with context hiding. *Black Hat EU*, 2016.

Knu98. Donald E Knuth. *The Art of Computer Programming: Sorting and Searching, volume 3*. Addison-Wesley Professional, 1998.

Koh04. 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.

KS00. 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.

KS12. Boku Kihara and Kazuki Shimizu. Considerations for Protocols with Compression over TLS. Internet-Draft draft-kihara-compression-considered-harmful-01, Internet Engineering Task Force, October 2012. Work in Progress.

KT14. James Kelley and Roberto Tamassia. Secure compression: Theory & practice. Cryptology ePrint Archive, Report 2014/113, 2014.

KZ16. Dimitris Karakostas and Dionysis Zindros. Practical new developments on BREACH. *Black Hat Asia*, 2016.

LGD15. Olivier Levillain, Baptiste Gourdin, and Hervé Debar. TLS record protocol: Security analysis and defense-in-depth countermeasures for HTTPS. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *ASIACCS 15*, pages 225–236. ACM Press, April 2015.

Lor18. Blake Loring. A solution to compression oracles on the web. https://web.archive.org/web/20260221083712/https://blog.cloudflare.com/a-solution-to-compression-oracles-on-the-web/, 2018.

LP18. Guillaume Lagarde and Sylvain Perifel. Lempel-Ziv: a "one-bit catastrophe" but not a tragedy. In *SODA*, pages 1478–1495. SIAM, 2018.

LS97. 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.

LZ4. LZ4. https://web.archive.org/web/20260312182047/https://github.com/lz4/lz4.

Met. Meta Platforms, Inc. Zstandard. https://web.archive.org/web/20250831013427/https://facebook.github.io/zstd/.

min. miniz. Miniz. https://web.archive.org/web/20260320152512/https://github.com/richgel999/miniz.

MK24. Marina Minkin and Baris Kasikci. ZipChannel: Cache side-channel vulnerabilities in compression algorithms. In *DSN*, pages 223–237. IEEE, 2024.

MvOV96. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, 1996.

Naf18. Ahamed Nafeez. Compression oracle attacks on VPN networks. *Black Hat USA*, 2018.

NNP19. Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. DiffFuzz: differential fuzzing for side-channel analysis. In *ICSE*, pages 176–187. IEEE / ACM, 2019.

PBP+17. Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In Boris Köpf and Steve Chong, editors, *CSF 2017 Computer Security Foundations Symposium*, pages 328–342. IEEE Computer Society Press, 2017.

PDM+18. 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.

PFSG22. 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.

PHG13. Angelo Prado, Neal Harris, and Yoel Gluck. SSL, GONE IN 30 SECONDS. https://web.archive.org/web/20260228024614/https://www.breachattack.com/, 2013.

PKW. PKWARE, Inc. PKZIP. https://web.archive.org/web/20260308163334/https://www.pkware.com/products/pkzip.

PR15. Roberto Peon and Herve Ruellan. HPACK: Header compression for HTTP/2. RFC 7541, 2015.

PRS11. 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, Berlin, Heidelberg, December 2011.

PS18. Kenneth G. Paterson and Jacob C. N. Schuldt. Statistical attacks on cookie masking for RC4. Cryptology ePrint Archive, Report 2018/093, 2018.

PSB12. Alfredo Pironti, Pierre-Yves Strub, and Karthikeyan Bhargavan. Identifying website users by tls traffic analysis: New attacks and effective countermeasures. Technical Report RR-8067, INRIA, 2012.

PSPW19. 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.

PST23. Kenneth G. Paterson, Matteo Scarlata, and Kien T. Truong. Three lessons from Threema: Analysis of a secure messenger. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1289–1306. USENIX Association, August 2023.

RD12. Juliano Rizzo and Thai Duong. The CRIME attack. *Ekoparty*, 2012.

RGA. Greg Roelofs, Jean-loup Gailly, and Mark Adler. zlib: A massively spiffy yet delicately unobtrusive compression library (also free, not to mention unencumbered by patents). https://web.archive.org/web/20260318094451/http://www.zlib.net/.

Ris13. Ivan Ristic. Defending against the BREACH attack. https://web.archive.org/web/20260223231743/https://blog.qualys.com/product-tech/2013/08/07/defending-against-the-breach-attack, 2013.

SBS+23. 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.

Sch96. Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, New York, second edition, 1996.

Sew. Julian Seward. bzip2 and libbzip2. https://web.archive.org/web/20250831013427/https://sourceware.org/bzip2/.

Sha49. Claude E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

Sha07. Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 552–561. ACM Press, October 2007.

Sih25. Oliver Sihlovec. Exploring decompression timing side-channels. Semester project, ETH Zurich, 2025. https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/semester-project_oliver-sihlovec.pdf.

Sna. Snappy. https://web.archive.org/web/20260311164412/https://github.com/google/snappy.

Son24. Yuanming Song. Refined techniques for compression side-channel attacks. Master's thesis, ETH Zurich, 2024. https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/masters-thesis_yuanming-song.pdf.

SS82. James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

Sul16. Nick Sullivan. CRIME, TIME, BREACH and HEIST: A brief history of compression oracle attacks on HTTPS. https://web.archive.org/web/20260123104311/https://www.helpnetsecurity.com/2016/08/11/compression-oracle-attacks-https/, 2016.

TMS+25. Kien Tuong Truong, Simon-Philipp Merz, Matteo Scarlata, Felix Günther, and Kenneth G. Paterson. Breaking and fixing content-defined chunking. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025*, pages 2294–2308. ACM Press, October 2025.

TSFS20. Po-An Tsai, Andrés Sánchez, Christopher W. Fletcher, and Daniel Sánchez. Safecracker: Leaking secrets through compressed caches. In *ASPLOS*, pages 1125–1140. ACM, 2020.

TV11. Cihangir Tezcan and Serge Vaudenay. On hiding a plaintext length by preencryption. In Javier Lopez and Gene Tsudik, editors, *ACNS 2011*, volume 6715 of *LNCS*, pages 345–358. Springer, Berlin, Heidelberg, June 2011.

VSN+21. Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. Opening Pandora's box: A systematic study of new ways microarchitecture can leak private data. In *ISCA*, pages 347–360. IEEE, 2021.

VVG16. Mathy Vanhoef and Tom Van Goethem. HEIST: HTTP Encrypted Information can be stolen through TCP-windows. *Black Hat USA*, 2016.

WBC+08. 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.

WBMM07. 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.

WBP+26. Ping-Lun Wang, Fraser Brown, Riccardo Paccagnella, Eyal Ronen, Riad S. Wahby, and Yuval Yarom. Transient architectural execution: From weird gates to weird programs. In Cristina Nita-Rotaru and Nicolas Papernot, editors, *2026 IEEE Symposium on Security and Privacy*, pages 1226–1242. IEEE Computer Society Press, May 2026.

Wil91. Ross N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference*, pages 362–371. IEEE Computer Society, 1991.

WPG+24. Yingchen Wang, Riccardo Paccagnella, Zhao Gang, Willy R. Vasquez, David Kohlbrenner, Hovav Shacham, and Christopher W. Fletcher. GPU.zip: On the side-channel implications of hardware-based graphical data compression. In *2024 IEEE Symposium on Security and Privacy*, pages 3716–3734. IEEE Computer Society Press, May 2024.

WPWB24. Ping-Lun Wang, Riccardo Paccagnella, Riad S. Wahby, and Fraser Brown. Bending microarchitectural weird machines towards practicality. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

YG19. Meng Yang and Guang Gong. Lempel-Ziv compression with randomized input-output for anti-compression side-channel attacks under HTTPS/TLS. In *FPS*, volume 12056 of *Lecture Notes in Computer Science*, pages 117–136. Springer, 2019.

Zie16. Michał Zieliński. SafeDeflate: compression without leaking secrets. Cryptology ePrint Archive, Report 2016/958, 2016.

ZL77. Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

zli. zlib-ng. zlib-ng. https://web.archive.org/web/20260309164003/https://github.com/zlib-ng/zlib-ng.

Zop. Zopfli. https://web.archive.org/web/20260314172247/https://github.com/google/zopfli.