debreach: Selective Dictionary Compression to Prevent BREACH and CRIME

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Brandon Paulsen

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Professor Peter A.H. Peterson

July 2017

Acknowledgements

First, I'd like to thank my advisor Peter Peterson and my lab mate Jonathan Beaulieu for their insights and discussion throughout this research project. Their contributions have undoubtedly improved this work. I'd like to thank Peter specifically for renewing my motivation when my project appeared to be at a dead–end. I'd like to thank Jonathan specifically for being my programming therapist.

Next, I'd like to thank my family and friends for constantly supporting my academic goals. In particular, I'd like to thank my mom and dad for their emotional support and encouragement. I'd like to thank my brother Derek and my friend Paul "The Wall" Vaynshenk for being great rock climbing partners, which provided me the escape from work that I needed at times. I'd like to again thank Jonathan Beaulieu and Xinru Yan for being great friends and for many games of Settlers of Catan. I'd like to thank Laura Krebs for helping me to discover my passion for academics and learning.

Finally, I'd like to thank my fellow graduate students and the computer science faculty of UMD for an enjoyable graduate program. I'd also like to thank Professor Bethany Kubik and Professor Haiyang Wang for serving on my thesis committee.

Abstract

Compression side–channel attacks like CRIME and BREACH have made compression a liability even though it is a powerful tool for improving efficiency. We present debreach, a step towards a general and robust mitigation for these attacks. A modified DEFLATE compressor with output that is fully backwards–compatible with existing decompressors, debreach has the ability to mitigate compression side–channels by excluding from compression sensitive data (e.g., security tokens, emails) identified either by explicit byte ranges or through string matching. In terms of usability, security, and efficiency, we find that string matching is well–suited to the task of protecting security tokens, but we also find that existing approaches to token security work equally as well. On the other hand, we find explicit byte ranges are well–suited to protect arbitrary content, whereas existing approaches lack in either efficiency or generality. When compared to the widely–used and insecure zlib in realistic scenarios, explicit byte ranges reduce throughput in networked connections by 16–24% on popular website's data, though this still results in a 106–269% improvement over not compressing depending on the available bandwidth. While the reduction is significant, we show that debreach can still improve throughput on connections between 112–208 Mb/s. We end with a discussion of practical use cases for debreach along with suggestions for their implementation and potential improvements to the algorithm.

Contents

List of Figures

1 Introduction

Compression works by identifying redundancies in an input and producing an output containing the same information but using less space. DEFLATE is a ubiquitous compression format that utilizes a technique called dictionary compression, which saves space by identifying repeated strings in the input and replacing them with references to a previous occurrence. In this way, repetitions of a string require very little space in the compressed output.

Simply knowing how well an input text compresses reveals information about the input without looking at the input itself. For example, if an input compresses well with a DEFLATE–conforming compressor, one can infer that the input has many repeated strings. Because encryption does not hide the size of the underlying plaintext, this information may be leaked in an encrypted connection if the plaintext is first compressed. Since 2012, several independent groups of researchers have shown that the information leaked by dictionary compression can be exploited to break encrypted connections when the data being transferred is first compressed [8, 5, 21, 12]. The proposed attacks can decrypt part of an encrypted message with time complexity that is linear or logarithmic relative to the amount of data being decrypted. The attacks have two requirements: the assailant can inject partially chosen plain texts into the message before compression and encryption, and the assailant can measure the size of the resulting ciphertext.

The proposed attacks have caused enough concern such that both TLS 3.0 and HTTP/2 have changed their specifications to avoid leaking information through dic-

tionary compression. TLS 3.0 has removed compression support entirely [20], and HTTP/2 has sacrificed compressibility in its header compression algorithm to close the leak [17]. However, the most common source of leakage – in HTTP body compression – remains largely unmitigated. Given that around 70% of websites compress HTTP bodies [22], this is concerning. Many solutions have been proposed for this leak, though they are lacking in either generality, performance, or security.

In this research, we work towards a solution for closing the leak in HTTP body compression that has performance comparable to its unprotected counterpart, guarantees security, and is more general than previous solutions. Our primary contribution is *debreach*, a DEFLATE-conforming compressor that has the ability to exclude selected parts of input data from dictionary compression, which effectively prevents leaking information about the selected data.

In the next section, we introduce prerequisites for understanding compression side–channel attacks and the most recent attack BREACH. Next, we describe the current proposed solutions in section 2.2, which serve as one point of comparison for debreach. In section 3, we cover the implementation details of debreach, and we present an example integration of debreach into the Apache–based web application phpMyAdmin, which has a known BREACH vulnerability. We then move on to benchmarking debreach in section 4, where we compare the performance of debreach to standard zlib, no compression, and the other proposed solutions where possible. In addition, we validate debreach's ability to exclude the selected data from compression before measuring debreach's performance. The procedures for validation are explained in sections 3.3 and 4.2. Finally, we draw conclusions from the performance results about the scenarios where debreach would be the optimal solution for compression–side channels in section 5. We also include potential future work in this section.

2 Background

2.1 Background

The primary purpose of this research is to understand and prevent compression vulnerabilities in a manner that still allows us to profit from compression. We focus on preventing the most recent and serious compression vulnerabilities which occur in HTTPS, so we open with an overview of the components of HTTPS – HTTP and TLS. Next, we cover the lower–level network protocols TCP and IP because some of their characteristics enable compression vulnerabilities. Then we describe in detail how compression introduces vulnerabilities into HTTPS, and we follow up with the two real–world compression side–channel attacks BREACH and CRIME. Finally, we close with a discussion of previously proposed mitigations.

2.1.1 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is not itself a protocol but rather the combination of two protocols: the Hypertext Transfer Protocol (HTTP) [6] and the SSL/TLS protocol [4]. Generally speaking, HTTP is a protocol for exchanging data between two hosts connected by a network. It operates using a request–response model where a client host makes a request to a server, and the server responds to the client with the results of the request, which could possibly be an error message if the request is invalid. Most often the client requests content from the server to display in a web browser. These requests are known as an HTTP GET requests.

For example, if a user wishes to access her web mail server, she enters the URL: *webmail.com* into her browser. When she hits the enter key, her web browser (the client) sends an HTTP GET request to the server running webmail.com to which the server responds with its home page. In addition, this page may contain instructions to request additional content from the server such as images. For example, there may be a reference to an image of the server's banner: *<img src="images/banner.png">*, which will cause the client's browser to send another HTTP GET request for *webmail.com/images/banner.png*. One final remark about the behavior of HTTP is in order before moving on to the structure of HTTP messages; in many situations, the server will include the client's request URL in the response. For example, if the client makes the request *webmail.com/images/eagle.png*, the server may include *webmail.com/images/eagle.png* somewhere in its response often for debugging purposes. As will be discussed, this behavior can enable a class of compression vulnerabilities.

Client requests and server responses are composed of two parts: a header and an optional body. The header contains information about both the client and server, the request URL, and metadata about the body. For example, the header indicates the compression method used on the body if any is used, the length of the body in bytes, and session cookies. The body is the data being transferred. In responses, the body contains the information requested by the client or an error message if the request failed. In requests, the body is used by the client to specify request parameters.

Secure Sockets Layer/Transport Layer Security (SSL/TLS) is a protocol for establishing an encrypted communication channel between two users. TLS is the successor to SSL which was deprecated in June of 2015 [1] due to inherent vulnerabilities, so we will only concern ourselves with TLS. TLS can perform a variety of tasks related to securing communication channels, but the ones relevant to this work are: verifying a user's identity, sharing a symmetric encryption key between two users in a

secure manner, and encrypting/decrypting the data sent between two users. Thus, when TLS is performed correctly, the users can be reasonably certain that they are not communicating with an imposter, and their connection will be unintelligible to an eavesdropper. Due to characteristics inherent in TLS, an eavesdropper can still transparently intercept and relay the unintelligible data, thus allowing the size of the data being transferred to be inferred. Inferring the size of the data is a key ingredient in enabling compression side–channel attacks such as BREACH. Like HTTP, TLS messages, called TLS records, have a header–body structure. The header contains metadata about the TLS connection, and the body contains the data being transferred. After the users have established an encryption key, only the TLS body is encrypted.

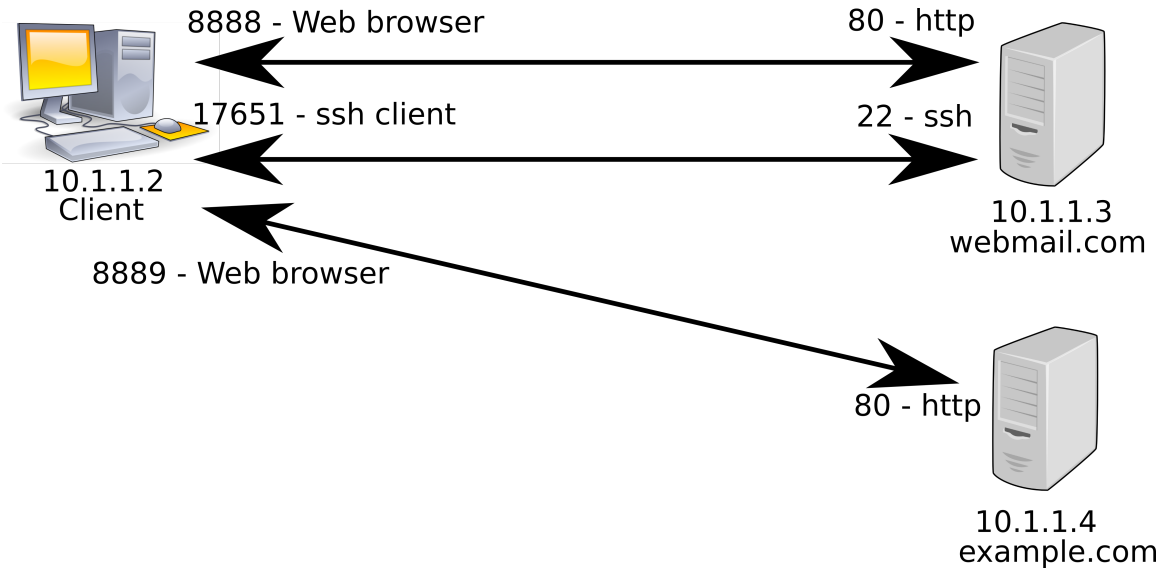

Figure 2.1: A high–level view of a client with several TCP/IP connections.

HTTPS operates on top of the Transmission Control Protocol (TCP) [19] and the Internet Protocol (IP) [18]. TCP provides reliable and uncorrupted delivery of data between processes which may or may not be running on the same host. Since a single host can have multiple processes and each process can have multiple connections,

TCP associates a port number with each side of each connection so the sending and receiving processes can be identified on their respective hosts. The concept of ports is illustrated in figure 2.1. The server webmail.com is running two processes: a web server on port 80, and an ssh server on port 22. The client has two processes running, a web browser and an ssh client, and the client has established three connections. To illustrate how a single process can have multiple connections, the client's web browser has two connections between two distinct processes each running on a distinct host.

IP differs from TCP in that IP is used for inter–host communication rather than inter–process communication. IP associates each host on a network with a unique, numerical identifier called an IP address as is shown below each machine in figure 2.1. To further clarify, a port identifies a process on a host, and an IP address identifies a host on a network. We can also observe from this diagram that each TCP/IP connection is identified by four values: a pair of IP addresses and a pair of ports.

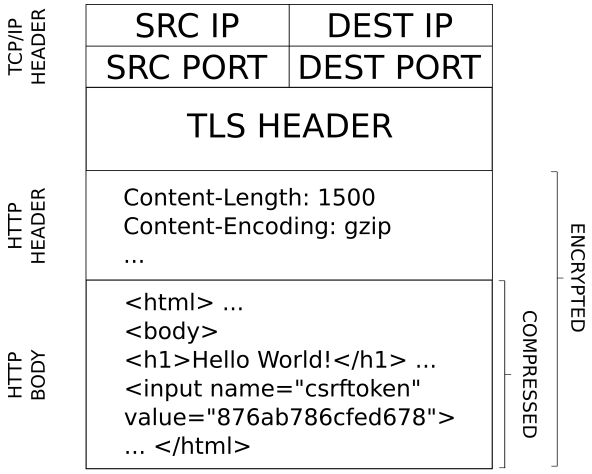

Figure 2.2: A high–level view of an HTTPS response packet.

Finally, putting these four protocols together, we can send an HTTPS message across the network. A high-level diagram of an HTTPS response is illustrated in figure 2.2. Notice the hierarchy of the message. Only the HTTP body is compressed when HTTP compression is used, so HTTP bodies are of special interest because we a primarily concerned with compression vulnerabilities. Next, notice that the HTTP message is contained within the TLS record body, so the entire HTTP message is

encrypted. Further note that the TCP/IP header is not encrypted so that the IP and port are still visible.

2.1.2 DEFLATE Compression

Compression techniques fall into two categories: lossy and lossless. Lossy compression reduces a file's size by irreversibly deleting non-essential data. Lossless compression reduces a file's size by encoding redundant data with fewer bits than its literal representation. The only implemented compression vulnerabilities occur in DEFLATE–conforming compressors which are always lossless. In this section, we give a brief overview of the building blocks of DEFLATE, namely LZ77 and Huffman coding, and we introduce terminology that will be used throughout this writing. In section 3.1, we dive into the source code of zlib, a widely–used library that implements a DEFLATE–conforming compression algorithm.

Huffman Coding

First presented by David Huffman [10], Huffman coding is a compression technique that encodes the bytes of a message as variable–length bit strings. Frequently occurring bytes are encoded using shorter bit strings whereas uncommon bytes are encoded using longer bit strings. Often the bit string used to encode uncommon bytes is greater than 8 bits, however this expansion is offset by the condensing of common bytes. Huffman coding relies on the existence of "common" bytes. A file with a near flat distribution of all possible 256 bytes will expand under Huffman coding.

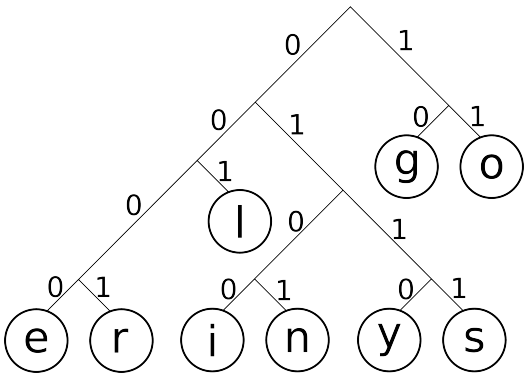

Figure 2.3: A Huffman tree

g = 10
o = 11
l = 001
r = 0001
n = 0101
i = 0100
e = 0000
y = 0110
s = 0111

Figure 2.4: Huffman codes

The encoding for each byte is stored in a data structure called a Huffman tree (shown in figure 2.3). A byte's encoding is determined by tracing the path from the root of the tree to the byte's leaf. The encoding is read one bit at a time. Following a node's left child indicates a "0" bit, and following the right child indicates a "1" bit. The Huffman codes for each byte as read from this tree are shown in figure 2.4.

LZ77

Originally proposed by Abraham Lempel and Jacob Ziv [25], LZ77 reduces redundancy by replacing a repeating string with a pointer to the first occurrence of the string. Say we want to compress the message "googling googly googlers". The string "googl" appears three times in this message, so we replace two of the strings with pointers to the first. Pointers are encoded as <*length, distance*> pairs, meaning read the *length* bytes that appeared *distance* bytes ago. In this case, the message compresses to "googling <9,5>y <7,5>ers". As will be discussed, this method of compression can allow encrypted communication to be decrypted in linear time.

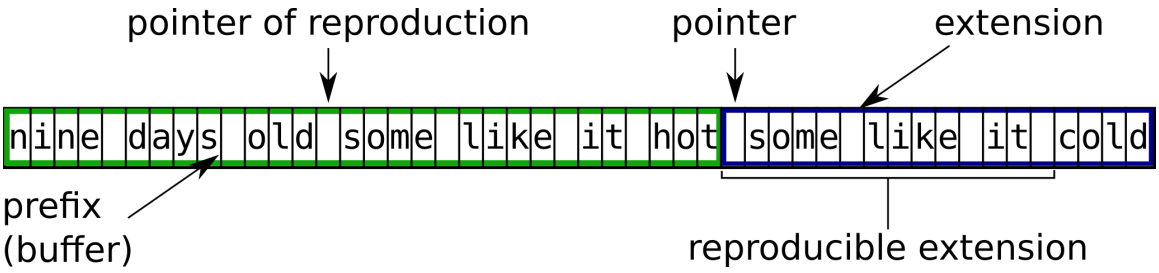

Figure 2.5: The sliding window as described by Lempel and Ziv

Lempel and Ziv give a formal definition and algorithm for their compression technique. The algorithm they describe is a brute–force, test–all–possibilities method. They use a pointer to divide the file into two parts. The area before the pointer is called the prefix, and the area after, including the pointer, is called the extension. They begin by setting the pointer to the beginning of the file. The prefix (which is empty in the beginning) is then searched in a brute–force manner for the longest string that matches the extension starting from the pointer. The portion of the extension that we match is called the reproducible extension of the prefix. The index in the prefix where the match begins is called the pointer of reproduction. If the length of a match passes a predetermined threshold, we write a length–distance pair to the compressed file, and advance the pointer to the index immediately past the reproducible extension. If no suitable match is found, we write the literal byte to the compressed file, and advance the pointer to the next byte. We continue in this fashion until the end of the file is reached.

Figure 2.5 depicts the compression algorithm mid-way through processing a text file that contains the nursery rhyme “Pease Porridge Hot”. A search of the entire prefix has returned the displayed pointer of reproduction for a match of length fourteen, including white space. An acceptable match is usually at least length three, so we output the length-distance pair *<14, 17>*. After this step, we advance the to the “c” in cold, so that the entire reproducible extension is now part of the prefix.

2.1.3 Compression Vulnerabilities

John Kelsey is credited with suggesting the first compression side–channel attacks on encryption in 2002 [13]. He describes two categories of attacks: passive and active. In the passive attacks, the assailant simply observes the compressibility of data. This has three requirements:

1. Data is compressed before it is encrypted
2. The attacker can infer the compressor's output length
3. The attacker, in some manner, can infer the compressor's input size

Requirement 1) and 2) are trivial because in practice data is always compressed before it is encrypted (encrypting first would render compression ineffective), and encrypted communication protocols do not hide the length of the underlying plaintext. Requirement 3) requires a special circumstance, such as malicious code on the victim's computer. As a result of Kelsey's passive attacks, the assailant may be able to determine the data type of the underlying plaintext. In addition, if the assailant also knows that one of a few different plaintexts will be encrypted, he maybe able to determine which plaintext was sent based on the observed compressibility. This fourth requirement is unrealistic in practice however. We do not pursue a solution to Kelsey's passive attacks for two reasons. First, an attacker that can infer the compressor's input size would likely have better attacks at his disposal than the attack described by Kelsey. Second, Kelsey states that these types of attacks are generally not devastating.

Kelsey describes a much more serious type of active attack that exploits the behavior of LZ77 compression to extract a string from a ciphertext. The active attack

has the first and second requirements of the passive attack, but not the third. It also requires that the attacker can inject a chosen plaintext into the target message before it is compressed and encrypted. The attack is performed as follows. Say we have an encrypted message M that contains a secret value S prefixed by the string "secret=", and we can cause M to be generated, compressed, and encrypted on demand with a chosen plaintext G (G is for guess) somewhere in M. The basis of the attack is that M will compress the best with LZ77 if G matches S. Say S is some type of session token, for example "abcd1234", so M always contains "secret=abcd1234". To begin the attack, we generate an M with each of the guess strings "secret=a", "secret=b", "secret=c", etc... for every possible character that can appear in S. Generally, the M containing the guess that matches S will compress better than all the other M's with incorrect guesses, so we take the M with the smallest size and assume that the guess in M matches S. In our example, the M with "secret=a" should compress the best. We then repeat the process, generating M with the guesses "secret=aa", "secret=ab", "secret=ac", etc... and taking the smallest M to have the correct guess again. We continue in this manner until all of S is guessed. Note that the execution time of this strategy increases linearly with the length of S.

In practice, this attack has many caveats and additional steps to the attack. The assumption that a correct guess will always compress better than an incorrect guess is not true in many situations. Kelsey and others suggest many ways to improve the accuracy of the attack [13] [9] [8]. Kelsey's initial experiment obtained a 70% success rate without his suggested improvements, so higher accuracy can likely be obtained. Thus, the attack is highly plausible.

BREACH and CRIME

Indeed, two instances of Kelsey's attack have been implemented in real world scenarios. The first was presented by Thai Duong and Juliano Rizzo at the 2012 ekoparty [5]. Their attack, dubbed CRIME, was able to extract session cookies from HTTP headers when the victim's connection had TLS compression enabled. Since TLS compression was scarcely used (and generally not necessary when it was used) the solution to CRIME was to disable TLS compression. Later, Yoel Gluck, Neal Harris, and Angelo Prado presented another instance of Kelsey's method in an attack called BREACH at the 2013 USA Black Hat conference [9]. Unlike CRIME, BREACH targets HTTP responses with HTTP compression enabled. HTTP compression is ubiquitous. As a result, disabling compression is not feasible as a general solution.

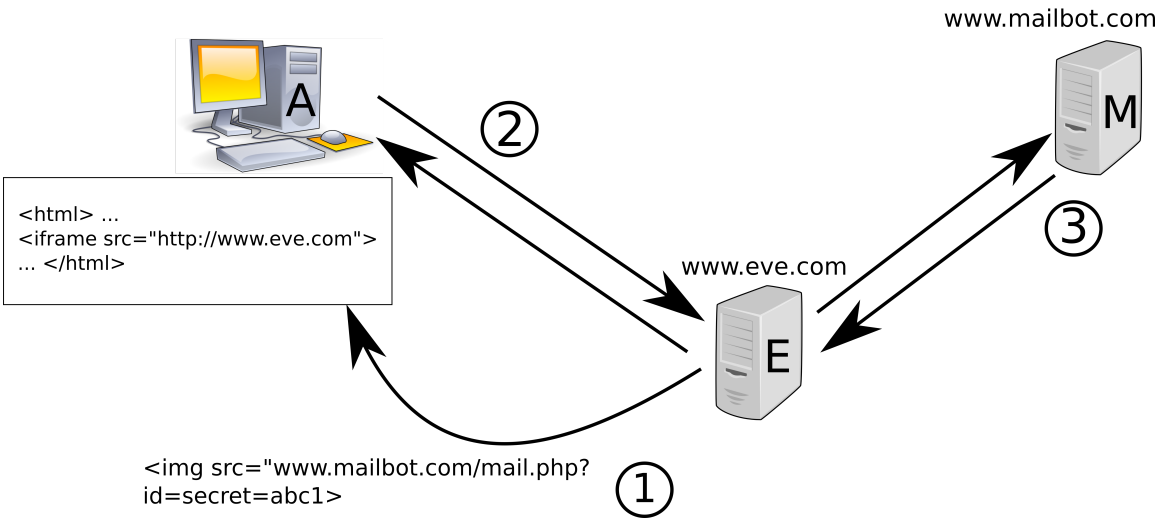

Figure 2.6: BREACH Topology

In their proof of concept, they attacked an encrypted webmail session between a victim Alice (A) and an Outlook mail server (M), as depicted in figure 2.6. The mail

server has HTTP compression enabled by default which is common practice. The target secret is a security token for the mail session which is always included in the HTTP body of the server's error responses. Their attack requires that Alice allows her mail client to automatically load HTML resources (e.g images and iframes) upon opening emails, and that the attacker Eve (E) can passively monitor the traffic between Alice and the mail server. Eve preludes the attack by sending Alice an email with iframe that references a webpage eve.com which she controls. The attack begins when eve.com receives a request, signaling that Alice opened the email. While Alice is distracted by Eve's colorfully–crafted email that promises large sums of money in exchange for helping a Nigerian prince, Eve begins injecting image tags into eve.com as shown in step one of figure 2.6. The image tag's source attribute is set such that it causes Alice's web browser to make a request to the mail server with a URL that contains Eve's guess at the secret value (step two). The request always fails though because "id=secret=..." is an invalid parameter for mail.php, and the mail server responds to Alice with an error message that contains both the security token and the bad request URL (step 3). Finally, Eve measures the size of this error message from her eavesdropping point. Thus, they recreate the scenario described by Kelsey; they have a message that contains a secret value, namely the error message and the security token, and they have a method of injecting chosen plaintext into the message before it is compressed and encrypted, namely sending a bad request. Finally, they have a method of measuring the compressor output. We implement our own proof–of–concept, which is discussed in section A.1.

Gluck, et al. suggest several mitigations for BREACH [8], the first of which is length hiding. By padding the ciphertext with random amounts of garbage data, the exact size of the compressor's output would be obscured. However, the exact size can still be determined if the same compressor output can be generated many times

by averaging the output sizes. This type of length hiding has been shown to only increase the time complexity of the attack from linear to logarithmic [12].

The second mitigation they suggest is masking sensitive secrets by XORing them with a random one-time pad which would be included with the secret. This means the attacker's injected guess would no longer cause the message to compress better if it matched the secret, thereby stopping the attack. Since the pad changes with each new message and it is random, the attacker cannot account for the masking. In this scheme, the secret remains the same across many requests, but its representation in each request would change. Masking has been effectively applied to protect security tokens in several web application frameworks (discussed in section 2.2.3). CTX is a framework that attempts to extend masking to arbitrary content, though its practicality is questionable (discussed in section 2.2.1).

The third mitigation they suggest is request–rate limiting. BREACH requires over a thousand requests to be made within a short period of time – more than a human would ever make. Like length hiding, request–rate limiting would slow the attack, but not entirely prevent it.

Fourth, they suggest requiring that the security token be presented for the server to respond to any client request. This too would work, however it may be extremely cumbersome to implement for large web applications, and it is prone to human error.

Finally, they suggest separating the compression contexts of user input and secrets. This could be done by serving the secrets in a response separate from all other content. While this would protect the data deemed "sensitive", this would introduce additional implementation complexities in the application, and anything contained in the same compression context as the user input would still be vulnerable. Instead, we explore a compressor–level method of separating compression contexts where the compressor avoids LZ77 matches that may be unsafe based on labeling the data's

source. One way we can automatically label the data's source is with a method called taint tracking.

2.1.4 Taint Tracking

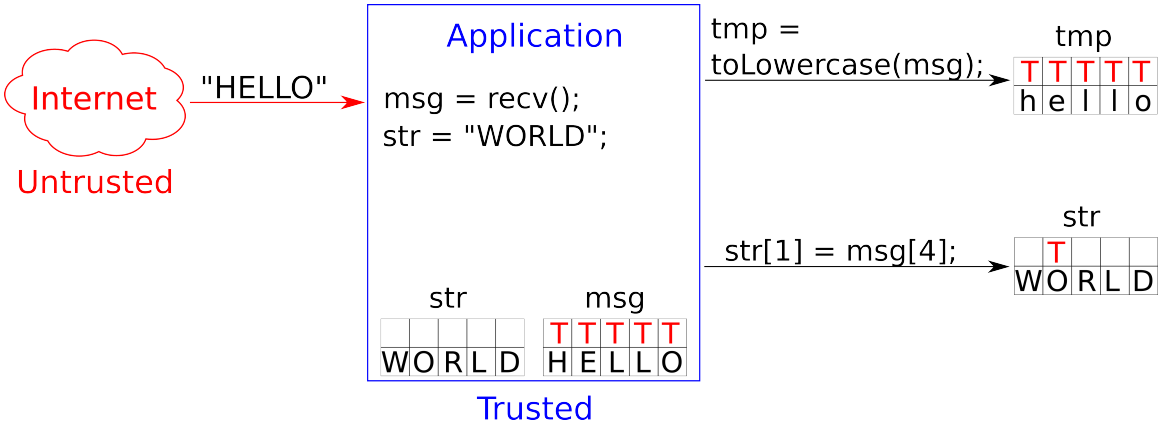

Figure 2.7: Taint Tracking Example

Taint tracking is a type of information flow tracking that focuses on how untrusted data is handled by enforcing a taint policy. Livshits describes the general components and taxonomy for taint tracking systems which nicely illustrates the workings of a taint tracking system [14]. Taint tracking systems have five components: sources, sinks, propagators, sanitizers, and a policy table. A source is an entity from which we receive untrusted data, for example, the *recv()* method in a C program. Usually anything that delivers user controlled input is a source. A sink is a breakpoint in an application where certain variables are analyzed for taintedness, for example, the *mysql_query()* function of PHP. If a sink detects taintedness, the corresponding policy is triggered. A policy is the action taken that specifies how tainted data should be handled. Generally, the action is either throwing an error or calling a sanitizer. A sanitizer is a method that transforms untrusted data into a form that can be safely used. Finally, a propagator is any function in the implementing language that should

transfer the taintedness from its input parameters to its output parameters.

Combining taint tracking with a compressor that can exclude selected data from dictionary compression is an interesting solution to compression side–channel attacks because it could theoretically perform well in terms of generality, usability, and runtime overhead whereas existing approaches lack in one or more of these areas. If all user input is excluded from dictionary compression, then compression side–channels are robustly mitigated for all data. In addition, such a system would provide protection with very little user intervention. Finally, such a system could achieve relatively small runtime overheads if taint tracking was already present. As we will show in section 4, selected data can be efficiently excluded from dictionary compression, and some languages already have taint tracking built–in.

Unfortunately, taint tracking in its current state cannot be used as previously described due to an issue discussed in section 5.3.4, however small modifications to the taint tracking systems would fix the discussed issue. We include additional information on taint tracking and discuss several experimental taint tracking systems in section A.3.

2.2 Previous Work

2.2.1 CTX

Karakostas, et al. describe a BREACH mitigation framework for web applications that was inspired by the idea of separating compression contexts proposed by the creators of the attack [11]. A developer uses their framework to tag data with a unique identifier called an "origin", which identifies where the data came from. For example, in a web mail application, email addresses could be used to identify origins, and then each email body would be tagged with the email address of the sender. Then, for each

origin, the framework generates a random, reversible transformation and applies it to all data with that origin. Each origin receives a different transformation, thereby preventing purposeful cross-compression of data with different origins. The data is tagged and transformed on the server-side, and the transformation is undone on the client-side with JavaScript.

CTX is simple, and it robustly mitigates BREACH attacks in a variety of circumstances. However, we find this solution lacking in a number of respects. In its current state, CTX cannot protect data in embedded JavaScript because it surrounds the protected data in * * tags. Attempting to protect such data will either break the syntax of the protected JavaScript, or prevent it from executing in the order originally intended by the developer. Both Gmail and Wikipedia place security tokens in embedded JavaScript, which suggests this would be a critical issue. We acknowledge that an anonymous JavaScript function could perform the unmasking on security tokens in embedded JavaScript without breaking syntax or execution order, but CTX does not support this. We also point out here that the inserted * * tags are not removed, and they change the structure of the DOM, which could possibly break the developer's web application.

A broader area where CTX's approach falls short is protecting data formats such as JSON or XML. CTX relies on JavaScript automatically executing to undo its transformations. However, unlike HTML with embedded JavaScript, JSON and other data-only formats cannot not be protected in this way because they have no standard way of containing instructions in order to automatically undo their transformation.

2.2.2 SafeDeflate

SafeDeflate is a DEFLATE–conforming compression algorithm that prevents LZ77 from leaking information about secrets composed of characters in a secret alphabet

specified by the user [24]. The algorithm enforces a constraint that prevents an LZ77 match from beginning or ending with a character in the secret alphabet, which effectively prevents a match from starting or beginning in a secret. BREACH requires that a correct partial guess at the secret can match with the whole secret, so the guess will always end with a character from the secret alphabet. The constraint prevents this match. The attacker can still attempt to guess the whole secret in a brute–force manner. When the secret is randomly generated, guessing the whole secret would take an exponential number of guesses, but if the secret has non–random characteristics, such as in an email address, then the attack complexity can be reduced. Furthermore, if the attacker has a few good guesses at the secret, for example they know that one of a few email addresses will appear in the encrypted message, then the attack complexity again becomes practical.

SafeDeflate has the advantage of being easy to use; the developer only configures the secret alphabet. Whether or not security of the secrets is guaranteed depends on the nature of the secrets. The constraint also comes at a high cost. The generality of the matching constraint means many profitable and secure LZ77 matches are not made, which results in a substantial loss in compression performance. According to their tests, their algorithm's compression ratio is 146% worse than DEFLATE even when they limit the secret alphabet to hexadecimal digits.

2.2.3 Token Masking Frameworks

Two web application frameworks, Django and Ruby on Rails, have security token frameworks that perform generation, masking, unmasking, and validation of security tokens automatically. Often the developer only needs to install and enable. Behind the scenes, the framework generates a one–time pad to mask the token for each new response it is used in. The framework then watches incoming requests for the masked

token, unmasking it when it comes across. Since the token's representation changes randomly with each response, an attacker cannot infer if their chosen plaintexts match the token. This type of approach is generally limited to tokens because masking most other data would break the view on the client's browser. Token masking frameworks perform well in terms of usability and overhead though they lack in generality. We find they are often preferable to a compressor–level approach which is further discussed in section 5.2.4.

3 Implementation

The goal of this research is to develop and evaluate a compressor–level solution to compression side channel leakage. In this section, we discuss the implementation of the compressor and the tools for analysis. We begin with a code trace of the original, unmodified DEFLATE algorithm of zlib in section 3.1, then discuss our modifications which make up debreach in section 3.2. We follow with a walk–through of our security validation tools in section 3.3, which includes small additions to the inflate algorithm of zlib and several scripts. Finally, we give an overview of our proof-of-concept attack, mitigation, and the details of integrating debreach into a web application built on the Apache web server and PHP in section 3.4.

3.1 zlib

zlib is a compression library that implements several DEFLATE-conforming compression algorithms. Those that use LZ77 have have several levels of compression which determine parameters that control the amount of effort put toward finding the best possible matches. zlib's algorithm does a single pass over the data being compressed during which it looks for matching strings and calculates byte frequency statistics. We discuss how zlib is used from the user's perspective, and then we continue with the data structures and the algorithms used for compression. Finally, we discuss our modifications to prevent information leakage.

The user interacts with the compressor through the API defined in zlib.h. Com-

pression happens through calls to *deflate()*, which takes two arguments: a *z_stream* structure and a *flush* value.

```
ZEXTERN int ZEXPORT deflate OF((z_streamp strm, int flush));
```

The *z_stream* structure is associated with an input buffer to be compressed and the output buffer to where the compressed data should be written. The *z_stream* is also associated with the compressor's internal state, however it is not visible to the user.

```
typedef struct z_stream_s {
    z_const Bytef *next_in;     /* next input byte */
...
    Bytef    *next_out; /* next output byte should be
                           put here */
...
    struct internal_state FAR *state; /* not visible
                                         to applications */
...
} z_stream;
```

Calling *deflate()* on a given *z_stream* will consume data in the input buffer, though the compressor may choose not to write to the output buffer in order to achieve maximum compression. The *flush* value is used to force the compressor to write to the output buffer.

The user can either work with the *z_stream* directly, or zlib can handle it. In the first case, the user must allocate space for the *z_stream* structure, the input buffer, and the output buffer. Then they must call *deflateInit()* or *deflateInit2()* on the *z_stream* to initialize its internal state. In the latter case, the user can execute *gzopen()* and *gzwrite()*. *gzopen()* opens a file for output and returns gzfile structure

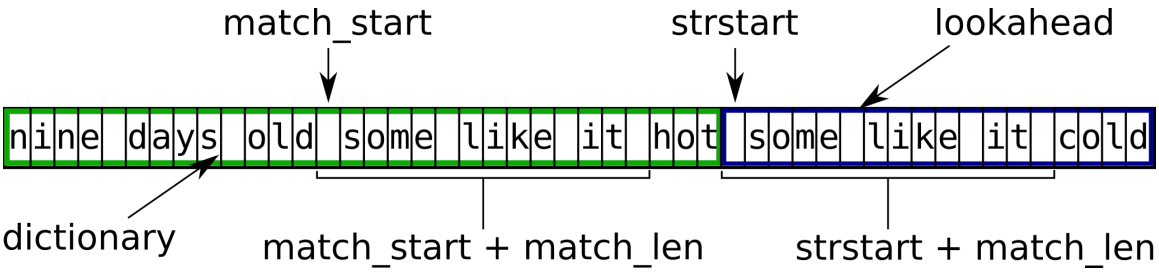

Figure 3.1: The zlib sliding window.

associated with the output file. *gzwrite()* takes a *gzFile*, input buffer, and a length l as arguments. It attempts to compress l bytes from the input buffer and writes the compressed data to the output file.

The *z_stream* holds another structure *deflate_state* which contains all variables associated with compression. Among those variables are those used for LZ77 compression, which we explain now.

LZ77 is often called "sliding window" compression because bytes from the data being compressed are sequentially read into a fixed size buffer (the window) contained in the *deflate_state*. When the buffer fills completely, bytes are dropped off the end to make room for more, thus it "slides" across the input. The algorithm sequentially processes each index in the buffer, and determines if a match can be found in previous data. If not, a literal byte is output, which is later Huffman coded. The data in the window before the byte being processed makes up the dictionary, and the data after the byte is called the *lookahead*. Figure 3.1 shows the case where a match has been found for the current index. *strstart* is the current window index we are processing, the *lookahead* is all data after *strstart*, *match_start* is the index of the match we found in the buffer, and *match_len* is the length of the match.

For each window index, we insert the key *window[strstart..strstart+2]* into a hash table mapped to *strstart*. In the case of collisions, values are either chained or overwritten depending on the compression level. Thus, for a given window index i, we can

quickly determine where previous occurrences of the string *window[i..i+2]* occurs in the window. Only indexes returned by the table look up are considered for matching during compression (this is a relevant detail in our modified algorithm). In addition, we must have enough lookahead, specifically the maximum possible match length, before testing a candidate match to ensure the best possible match is found.

3.2 debreach

3.2.1 General Approach

We mitigate attacks by excluding specified data from dictionary compression. The excluded data can be specified either by absolute positions in the buffer or by literal strings we wish to exclude (e.g., security tokens). The implementations for these two approaches substantially overlap; we discuss the overlap first.

Both implementations utilize metadata about each individual byte in the sliding window. We store the metadata in an array of 16-bit unsigned integers called *next_taint*; each *next_taint[i]* contains the metadata for *window[i]*. This requires that *next_taint* has the same size as the sliding window. The value at a given index indicates the number of bytes until the next exclusion zone. In general, the next index in the window that is part of an exclusion zone is i + *next_taint[i]*, except when the next zone is out of matching range for the given window index. In the latter case, the value is the maximum match length plus one. The technique for filling out this array differs between the two implementations, and it is discussed in the following two sections. Using this array, we can identify the areas in the sliding window that must be excluded from dictionary compression to ensure security.

In DEFLATE algorithms, there are four scenarios where dictionary compression might try to compress data in an exclusion zone. We modify zlib's compression algo-

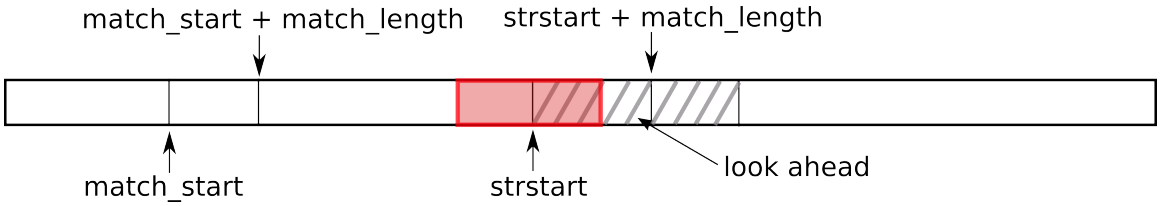

Figure 3.2: First scenario: The sliding window index (strstart) is in an exclusion zone, which is marked with red. The available lookahead is marked in grey.

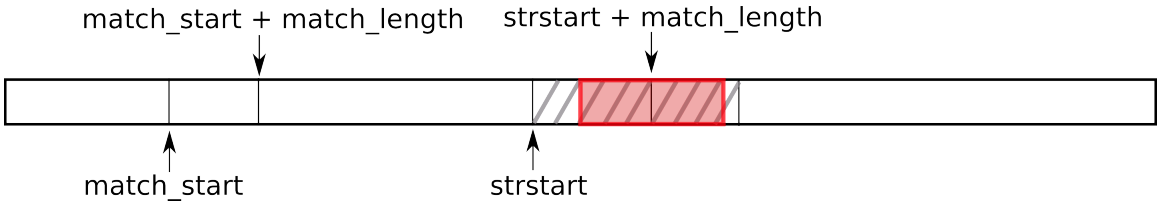

Figure 3.3: Second scenario: The reproducible extension extends into an exclusion zone.

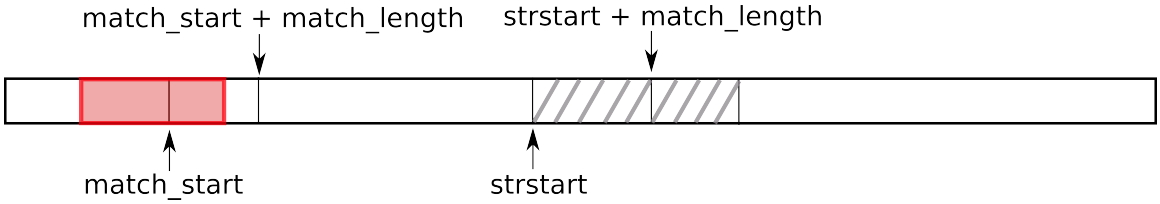

Figure 3.4: Third scenario: A match begins in an exclusion zone.

rithm to check for and handle each of these scenarios. We differentiate the scenarios based on two binary characteristics:

1. Whether it is the reproducible extension or the match that intersects an exclusion zone

2. Whether the intersection starts inside or outside an exclusion zone

In the first scenario, shown in figure 3.2, the reproducible extension intersects an exclusion zone, and the intersection begins inside the exclusion zone. We prevent intersections from beginning in exclusion zones (the first and third scenarios) before any matches are attempted. For the first scenario, we check that the current window index is not in an exclusion zone. We also consider the two bytes before an exclusion zone to be part of the zone itself because we cannot find a "good" match within two

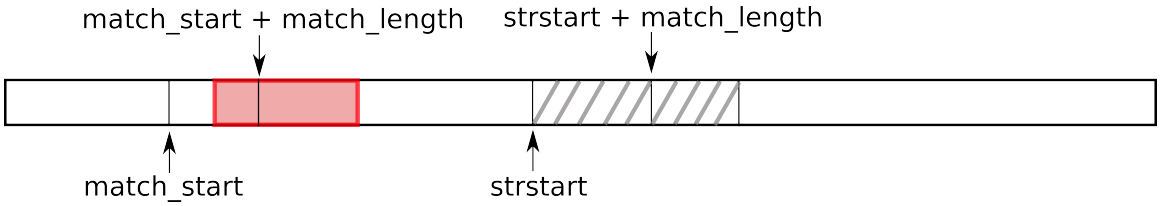

Figure 3.5: Fourth scenario: A match extends into an exclusion zone.

bytes of a zone. The magic number two comes from the minimum match length minus one. In code, this is the check:

```
if (next_taint[strstart] < MIN_MATCH)
```

When this check succeeds, we Huffman encode bytes until the same check evaluates to false. In addition, we do not insert these strings in the hash table, which has the effect of preventing the third scenario (figure 3.4) where a match begins in an exclusion zone.

We handle the second (figure 3.3) and fourth (figure 3.5) scenarios in the matching algorithm. In these cases we need not worry that the window index or a candidate match will begin in an exclusion zone, but that the reproducible extension or the match may extend into one. For the reproducible extension, we reduce the maximum match length so that it cannot extend into the zone. In code:

```
max_match =
    next_taint[strstart] > MAX_MATCH ?
    MAX_MATCH : next_taint[strstart];
```

The match requires a more involved technique because the distance to the next exclusion zone changes with each match we test. For each candidate match, we check if the first byte of the next exclusion zone (from the perspective of the match) matches with the corresponding byte in the lookahead. If so, the match could potentially extend into the exclusion zone. To stop this, we temporarily change the first byte of the exclusion zone in the window to guarantee that the matching will break before it

extends into the zone. Then we undo the change after the matching is complete. In code, the check and the change looks like:

```
next_taint_i =
    match_start + next_taint[match_start];


if (window[next_taint_i] ==
    window[strstart + next_taint[match_start]])
{
    window[next_taint_i]++;
}
```

3.2.2 Byte Range Based

We provide a client function for declaring unsafe data. The function, *taint_brs()*, takes a list of relative byte ranges which should be excluded from dictionary compression. The byte ranges are relative to the amount of data already passed to the compressor. For example, suppose a file of size 5,000 bytes is being compressed and we wish to exclude the last 1,000 bytes from compression. If we have passed 2,000 bytes to the compressor, we would declare the byte ranges 2,000 to 2,999 to *taint_brs()*. If no input has been given, then the byte ranges would be absolute positions. The compressor maintains an internal list of excluded byte ranges, which it eventually translates into values for *next_taint*.

3.2.3 String Based

Range based exclusion is more efficient, but is less foolproof because it requires more effort and "bookkeeping" on the part of developers to make sure that the byte

ranges are properly identified. String based exclusion removes the developer overhead of providing byte ranges but adds computational overhead for searching and identifying matches.

For string based exclusion, we maintain a list of strings in the compressor's internal state. We call these strings "unsafe". We exclude only entire matches of these strings; partial matches are allowed. We provide a function for the client, *declare_unsafe()*, which takes a list of strings and adds them to the internal "unsafe" list. Once a client adds strings to the list, future input will be matched against it.

The lookahead in the sliding window is searched each time new data is added to it. During each search, we set as many values in *next_taint* as possible. For a given window index i, we can set *next_taint[i]* if and only if one of the following conditions are met:

1. We can confirm that no unsafe strings begin at the indexes between i and $i + MAX\_MATCH$

2. We have confirmed an unsafe string occurs at an index greater than i

In addition, we often encounter a partial match of an unsafe string at the end of the lookahead. In this case, we force the compressor to wait for additional lookahead before allowing compression to continue unless the client flushes the compressor. Generally, the client only flushes the compressor when all data has been passed to the compressor. (Flushing more than once during compression is unusual and could harm security.)

3.3 Validation Tools

In this section we explain the implementation of the tools used for validation. The validation procedure is discussed in section 4.2. Validating the security of the compressor requires us to show that tainted byte ranges or unsafe strings never participated in dictionary compression. To do this, we analyze the dictionary matches output by the compressor and ensure they do not intersect any tainted regions. We modify the zlib decompressor to fit this purpose. During decompression we convert each match in the output file to a pair of regions — one for the reproducible extension and the other for the match location. Given the number of bytes b already output by the compressor and a match of the form *<length, distance>*, the region of the reproducible extension is b to $b + length - 1$, and the match range is $b - distance$ to $b - distance + length - 1$.

Finally, we compare the byte ranges from the decompressor to a gold standard of tainted regions. Determining the gold standard is trivial and depends on the experiment. (The procedure is discussed in section 4.2.) The problem then becomes: given two lists *l1* and *l2* of byte ranges, determine if any of them overlap. We do this using brute force, comparing all to all in a manner as such:

```
for each range1 in l1:
    for each range2 in l2:
        if overlap(range1, range2):
            return True
return False
```

3.4 Proof of Concept

We implement a proof of concept usage for debreach in the web application phpMyAdmin, which has a known vulnerability to BREACH [23]. phpMyAdmin is written in PHP and is often run via the Apache web server. Compression and PHP are integrated with the web server using its module framework. We first introduce the web server and the module framework, followed by the PHP and compression modules, the latter of which includes the integration of debreach. Finally, we show that data is correctly excluded from dictionary compression using our validation tools.

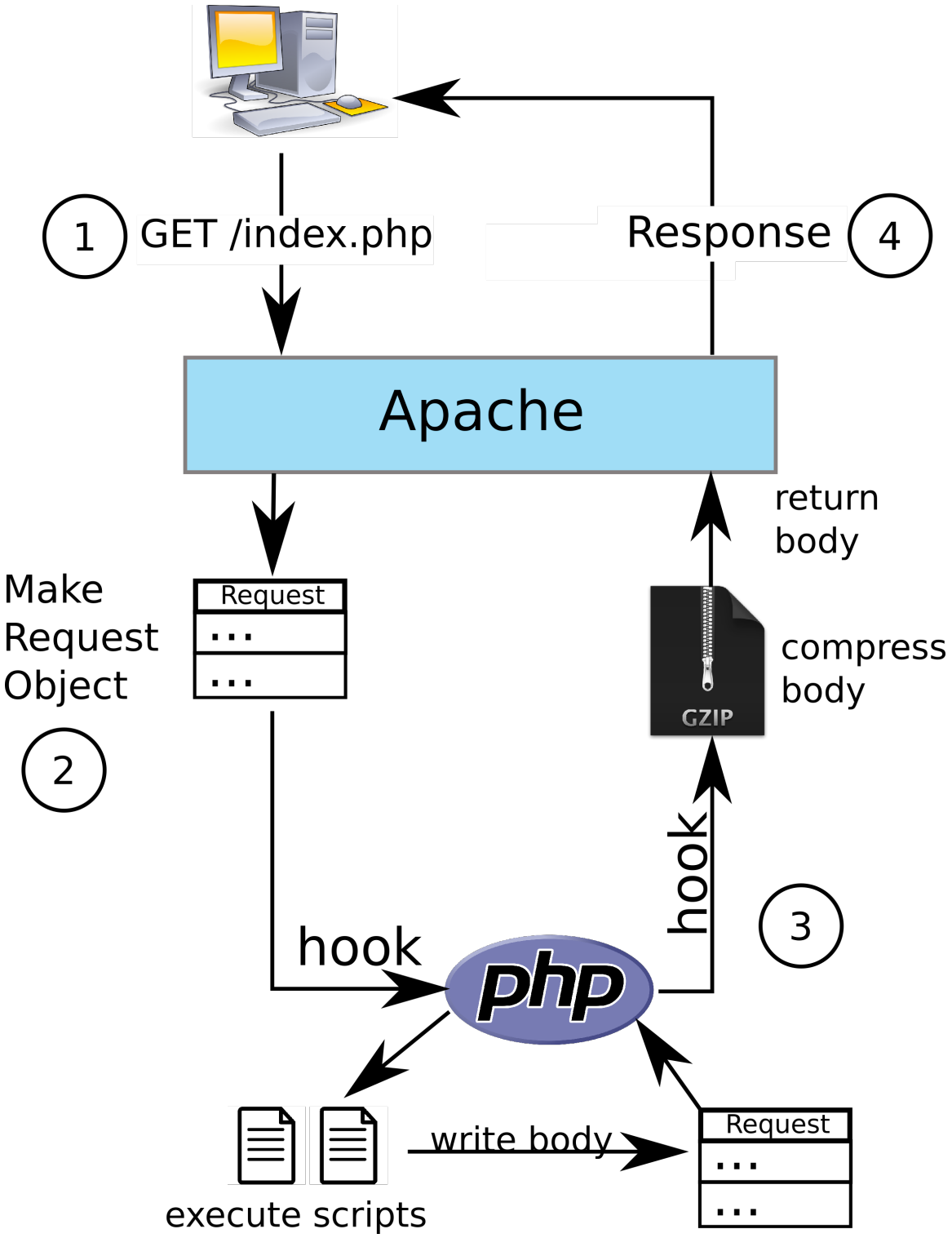

Figure 3.6: The life cycle of Apache handling a request. 1) Apache first reads the request from the client, then 2) constructs a request object and 3) passes it through the module chain. 4) The resulting body after passing through the chain is then returned to the client.

3.4.1 Apache and Apache Modules

Without modules, Apache has very little functionality; essentially it operates as an HTTP file server. Additional features, such as generating content through server-side scripting languages and compression, are provided by modules. Apache's implementation divides request processing into several phases where a modules can execute. Each module registers with Apache at least one function, called a "hook", to be called during one of the phases. For example, PHP executes during the request handling phase because its scripts create the content of the response. The compression module executes during a post handling phase because it compresses the final response content before it is sent to the client. The sequence of modules executed through the request's life cycle is called the *module chain*. The module chain shown in figure 3.6 and for our setup contains the PHP module and a compression module.

The function used to register hooks is one of many provided by Apache's server application programming interface (SAPI). The SAPI also provides functions that give total control of request handling, such as modifying the response headers and body. The hooks and the SAPI act on request objects which contain the data associated with the request made by the client and the response to be sent. Each request received from a client is converted to request object, and then passed down the module chain, so that each module has a chance to act on the request. The response body of the request object is returned to the client after passing through the chain. The request object looks something like figure 3.7, and it is shown being passed through the module chain in figure 3.6.

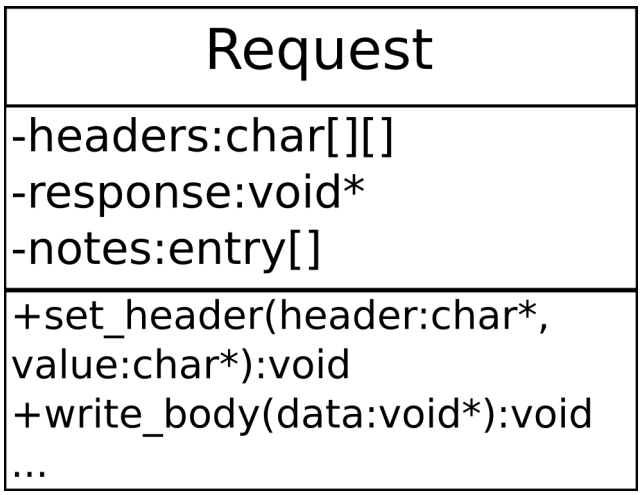

Figure 3.7: An example of a request object. It contains data members for the response headers, body, and request notes.

Another feature of Apache modules important to the proof of concept is inter-module communication. Since the compression and PHP modules are separate, we use this feature to to communicate security tokens generated by the web application to the compressor. The feature, called a "note", allows a module to create a key-value pair associated with the current request object as shown in figure 3.7. The PHP module provides a built-in PHP function *apache_note()* that allows creation of notes. In phpMyAdmin, we call *apache_note* each time a security token is generated, like so:

```
apache_note("debreach_unsafe_token", $security_token);
```

which gives the compression module knowledge of all security tokens in the incoming content.

mod_php

PHP is a stand-alone, server-side scripting language, but it is generally used in tandem with a web server to generate dynamic web pages. *mod_php* is the module that enables Apache to execute PHP scripts. *mod_php* can be broadly divided into

two parts: the server interfacing components and the PHP interpreter. The server interfacing components, which are essentially just calls to the SAPI, are primarily used by the PHP module to return the output of PHP scripts, but it also includes the functions for registering hooks and setting HTTP response headers. The handler hook "handles" the client's request by executing the scripts necessary for building the response.

mod_deflate

mod_deflate is the default and ubiquitous compression module included in the standard Apache package. Content is passed through this module as it is generated by the request handling module. It first performs several checks to decide whether or not compression can be profitable. Specifically, it checks that the benefit of compression is greater than the additional HTTP/gzip headers that are required when compression is used, and whether the incoming content is already compressed. After passing the checks, it reads as much content as it can into the compression buffer, and outputs compressed data as it sees fit. The compressor may wait for additional input in order to achieve better compression, except in the case when it receives an "end of output" signal from the request handling module.

phpMyAdmin

phpMyAdmin is web application built on PHP that provides a graphical user interface for administering databases running on the server. We use it in our proof of concept because it contains user–controlled data in certain HTML pages that contain security tokens, which makes it vulnerable to BREACH. Security token generation

and handling is implemented in the application source code, which is ideal for proof of concept.

3.4.2 *mod_debreach*

The debreach module that we introduce for Apache is nearly identical to *mod_deflate* except for a few function calls before the main compression routine is executed. Before compressing, we read the request object's notes for any new incoming security tokens. If new tokens are found, we declare them unsafe through debreach's API with a call to *declare_unsafe()*, which prevents the token from being compressed.

3.4.3 Integration into phpMyAdmin

phpMyAdmin handles security token generation in a single function *PMA_secureSession()*, and it stores the token using the built in session framework. Only one token is generated per session, which gives us two cases where we need to mark tokens for the compressor:

1. A new token is generated
2. A session is resumed

For new tokens, we set a note in the *PMA_secureSession()* function after the token has been created. Because phpMyAdmin uses the built-in session framework of PHP, we know where in the code a session is resumed because it must done by calling the function *session_start()*. After each call to this function, we pull the security tokens from the session framework and set notes for them. Both of these steps are done before any content is generated, so it is guaranteed that the tokens will not be compressed.

4 Results

4.1 Test Data

We test debreach on five sets of real-world HTTP traffic. Each set contains traffic captured from a single website during approximately ten minutes of human-driven browsing. The five websites chosen are Facebook, Reddit, Wikipedia, Gmail, and phpMyAdmin. During each browsing session, we save each HTTP response into a separate file and perform decompression where applicable, which gives a set of files for each site. In addition, we save the HTTP response headers for each file because they contain important information such as the response encoding — that is, was the response compressed — and the response type — for example, JavaScript, HTML, JSON, jpeg, etc. Facebook, Reddit, Wikipedia, and Gmail are public–facing websites that can be browsed directly. On the other hand, phpMyAdmin is designed to connect to a database on an internal network, so we create our own server running a database populated with a large amount of dummy data and an instance of phpMyAdmin. When browsing each site, the goal was to click on as many unique links as possible, but with preference towards links that a typical user would likely click as well.

Each of the chosen websites has unique characteristics, the most important being the number of tokens embedded per response. Gmail and Reddit rarely embed more than one security token in their responses, which makes them a best case for string debreach. In addition, both are text-heavy in the content they serve. Facebook and

Wikipedia usually embed between two and ten tokens per response, making them an average case for string debreach. Facebook differs from the rest in that it is image-heavy compared to the other sites. Finally, phpMyAdmin often embeds more than fifty tokens per response with a maximum of 2,040 tokens, making it a worst case. In addition, its content contains many long, repeated URLS, making it extremely compressible compared to the other sites.

For further examination, we generate an additional set of "worst case" data composed of approximately equal proportions of whole security tokens, partial security tokens, and random content pulled from the five sets of real–world data. The method for crafting this set results in about one–third of the data being tainted (when tokens are declared unsafe) and it will be extremely redundant. Not only does this give a worst case scenario for performance, but it also proved to be effective at finding bugs during validation testing. We craft 120 "worst case" files of 100KB each.

4.2 Validation Testing

Before performance testing, we confirm that debreach works as intended using the tools described in section 3.3. The important aspects to validate are integrity and security; that is, whether the compressor outputs DEFLATE–conforming files and whether the compressor correctly excludes the indicated byte ranges or strings from dictionary compression. Integrity can be validated by decompressing the output with any unmodified gzip–compatible compression utility. We use the gzip utility included with the unmodified zlib [7]. Security is more difficult to validate due to the complexity of monitoring the contents of the internal dictionary. We considered either monitoring the dictionary matching process during compression or analyzing the matches output by the compressor. We opted for the latter due to simplicity.

We use the decompressor described in section 3.3 to compare the matches encoded in the output to a gold standard of tainted data determined by analysis scripts separate from the compressor.

The process for validating a single file has four steps:

1. Before compression, record a gold standard of byte ranges that must be excluded
2. Perform compression with debreach
3. Decompress and save the match byte ranges to a file
4. Compare the gold standard byte ranges to those output by the decompression step

The method for step one depends on whether we are validating the byte range or string based approach. For byte ranges, we randomly generate a set of tainted regions for each file because the actual taintedness is not known. The regions in this step cover approximately 10% of the file. For strings, we first find all tokens present in each file either by hand or with scripts, and then we calculate the absolute byte ranges of the tokens. All six sets of data were run through the validation process and passed.

4.3 Performance Testing

We focus on three important measures in our comparisons: compression ratio, consumption rate, and effective throughput. Compression ratio quantifies the reduction in size achieved by compression. Smaller compression ratios indicate greater reductions in size, though smaller compression ratios don't necessarily indicate a "better"

performance. Formally:

$$compression \quad ratio = \frac{compressed \quad size}{original \quad size} \tag{4.1}$$

So, if a file with size 100KB compresses to 50KB, the compression ratio is 0.5. Compression ratio depends on both the compressibility of the data itself, and, in DEFLATE compressors, the algorithm's ability to find good matches. We use compression ratio to measure the loss in compressibility due to excluding regions of data from dictionary compression.

The consumption rate measures the speed at which the compressor consumes input. Higher consumption rates for compressors generally indicate simpler and faster, but not necessarily better implementations. Formally:

$$consumption \quad rate = \frac{input \quad size}{algorithm \quad execution \quad time} \tag{4.2}$$

We use consumption rate to measure the execution time overhead caused by the additional processing steps of debreach.

Compression ratio and consumption rate are usually at odds with each other. With DEFLATE, better compression ratios are achieved by implementations that spend more effort searching for quality matches. Conversely, better consumption rates are achieved by implementations that spend less effort on matching. Like the compression ratio, consumption rate also depends on the compressibility of the data. Finding a match means that the compressor can skip several processing steps specific to Huffman coding for the reproducible extension of the match. Our results show that zlib's default compression algorithm achieves significantly higher consumption rates on data sets with highly–redundant strings as compared to others. Consumption rate also depends on many aspects of the algorithm's implementation. In our case, the

only implementation details that vary pertain to the matching algorithm, and the rest is held constant.

While the consumption rate and compression ratio allow us to compare specific details of compression algorithms, they do not lend much insight into the real–world benefits of compression. The effective throughput provides that insight. Compressing data before sending it across a network is beneficial because it can increase effective throughput — the rate at which *information* is sent across a network. This is different from regular throughput, which is the rate at which *bits* of data are sent across a network. For example, suppose we have an 8,000 bit (1,000 byte) file that compresses to 800 bits (100 bytes), and the compressor outputs the compressed file at 400 bits/sec. With an available bandwidth of 400 bits/second and using compression, we would send this file across the network in 2 seconds, which gives us a throughput of 400 bits/sec and an effective throughput of 4,000 bits/second. Without compression, the file would send in 20 seconds, which gives us 400 bits/second for both measures. We've effectively increased our bandwidth ten–fold in this case. Note that the rate at which the compressor outputs bits can bottleneck the transmission. For example, suppose the available bandwidth was 8,000 bits/sec. With compression, we would still send the file across the network in two seconds for an effective throughput of 4,000 bits/sec. Without compression, the file would send in just 1 second for an effective throughput of 8,000 bits/sec. Thus compression is not always a win.

As our example shows, the effective throughput depends on the available bandwidth, the compressed size, and the speed of the compressor. We calculate effective throughput based on *compressed output rate* and *compression gain*. Compression gain (CG) is the inverse of compression ratio. Formally:

$$CG = \frac{original \quad size}{compressed \quad size} \tag{4.3}$$

Unlike compression ratio, larger compression gains indicate better compression performance.

Compressed output rate (COR) is like consumption rate, but in terms of compressed size rather than original size. Formally:

$$COR = \frac{compressed \quad input \quad size}{compression \quad time} \tag{4.4}$$

Like consumption rate, larger compressed output rates are not always better. Theoretically, no compression will achieve the highest compressed output rate because it requires the least amount of processing, but we just illustrated an example where no compression performs worse compared to compression.

Finally, the effective throughput (ET) is:

$$ET = min(ABW, COR) \times CG \tag{4.5}$$

where "ABW" is the available bandwidth. We consider no compression to achieve maximum COR and a CG of 1.0, which results in an ET equal to the ABW. Larger effective throughputs are always better. We use effective throughput to present two important aspects of debreach: when debreach is the optimal and secure choice, and the difference in overall performance of debreach as compared to other compression methods (including no compression).

We measure compression ratio, consumption rate, and effective throughput in two scenarios: with tokens tainted and with arbitrary content tainted. We do not test string based tainting in the second scenario because, as is discussed in section 5.1, this method is not fit for protecting arbitrary content. We also include tests to compare performance to CTX and SafeDeflate, alternatives to debreach.

4.3.1 Token Tainting

Compression Ratio

We measure the impact that token tainting would have on compression ratio for each data set compared to the standard, unsafe zlib. To make the tests more realistic, we remove files from each data set that were not compressed by the website's server according to the HTTP headers gathered during data collection. The ratios achieved by zlib and debreach are summarized in section 4.1. Note that even though we have two implementations of debreach (byte range and string), the compression ratios would be the same for both because the same data is excluded from dictionary compression for this experiment. In this case, the ratio is the total size of all files in the data set divided by the total compressed size. Loss is shown as the difference between the compression ratios achieved by zlib and debreach.

site	zlib	debreach	loss
Facebook	0.21324	0.21332	0.00008
Reddit	0.29013	0.29016	0.00003
Wikipedia	0.25677	0.25684	0.00007
Gmail	0.21432	0.21433	0.00001
phpMyAdmin	0.09427	0.11948	0.02521
crafted	0.08257	0.42401	0.34144

Table 4.1: Compression ratios for zlib and debreach, and the difference in compression ratio due to token exclusion.

The compression ratio losses are as we would expect; the loss increases with the number of tokens that a site embeds, and the loss is small when the number of tokens is small relative to the total content size. For all real sites except phpMyAdmin, we see nearly insignificant loss in compression ratio (less than 1/100th of a percent). As previously mentioned, phpMyAdmin is a worst case due to the large proportion of security token content. To elaborate, in the file that contains 2,040 tokens, standard zlib would replace 2,039 of these tokens with references that are approximately

three bytes each. On the other hand, debreach excludes the tokens from dictionary compression for security and only Huffman encodes them. This still compresses the input, but where zlib compresses the input to about 9.4% of the original size, debreach compresses to about 12%. This suggests that debreach's compression performance is strong in the token tainting scenario, even in realistic worst–case scenarios.

We include compression ratios achieved on the crafted data set to illustrate the pathological case for debreach and the importance of dictionary matching in achieving good compression ratios. The crafted data set contains *extremely* redundant data of which approximately one–third is tainted. This shows debreach's weakness; the compression ratio loss increases proportionally with both the amount of tainted data and the compressibility of the tainted data.

Finally, we measure the compression ratio achieved by Huffman–only compression (results shown in table 4.2) because it is a secure alternative to debreach. The results show that debreach significantly improves the compression ratio on all data sets and between 37 – 54% on the real–world data. Even though Huffman–only compression performs relatively poorly on compression ratios, it achieves much higher consumption rates because it avoids the complexity of finding matches, which is described in section 4.3.1.

site	Huffman	gain
Facebook	0.67082	0.45659
Reddit	0.66991	0.37975
Wikipedia	0.66276	0.40592
Gmail	0.66832	0.45399
phpMyAdmin	0.66460	0.54512
crafted	0.64132	0.21731

Table 4.2: Compression ratios for Huffman–only zlib compression and the differences between them and debreach. "loss" changes to "gain" because debreach performs better than Huffman coding.

Consumption Rate

We measure the consumption rate in the same first scenario as in the compression ratio experiments: tainting tokens in data that was compressed by the server. We make the assumption that the compressor knows all tokens that could possibly appear in the incoming data, and that tokens are declared unsafe regardless of if they appear in the incoming content. Alternatively, we could only declare tokens unsafe if they appear in the incoming content, but this would be an unrealistic benefit for the performance of string based debreach. We measure the time it takes to pass an entire test set through the compressor and divide the total size of the test set by the total run time to obtain the consumption rate in MB/s. For each test set and each algorithm, we repeat this process 60 times to obtain a distribution of consumption rates. A graphical comparison of the average consumption rates are shown in figure 4.1. The numerical averages and difference in consumption rates between debreach and zlib are shown in tables 4.3 and 4.4.

Compression was performed using a bare–bones command line utility that did nothing more than read in a file's contents and apply the compression algorithm. The utility still wrote the compressed data to an output buffer, but it did nothing with it so as to eliminate write overhead. The experiments were run on an Ubuntu 14.04 server with no GUI. All networking and non-essential processes were killed during the course of the experiment. The machine had a 2.93 GHz Intel Core i3-530 processor with 4GB RAM.

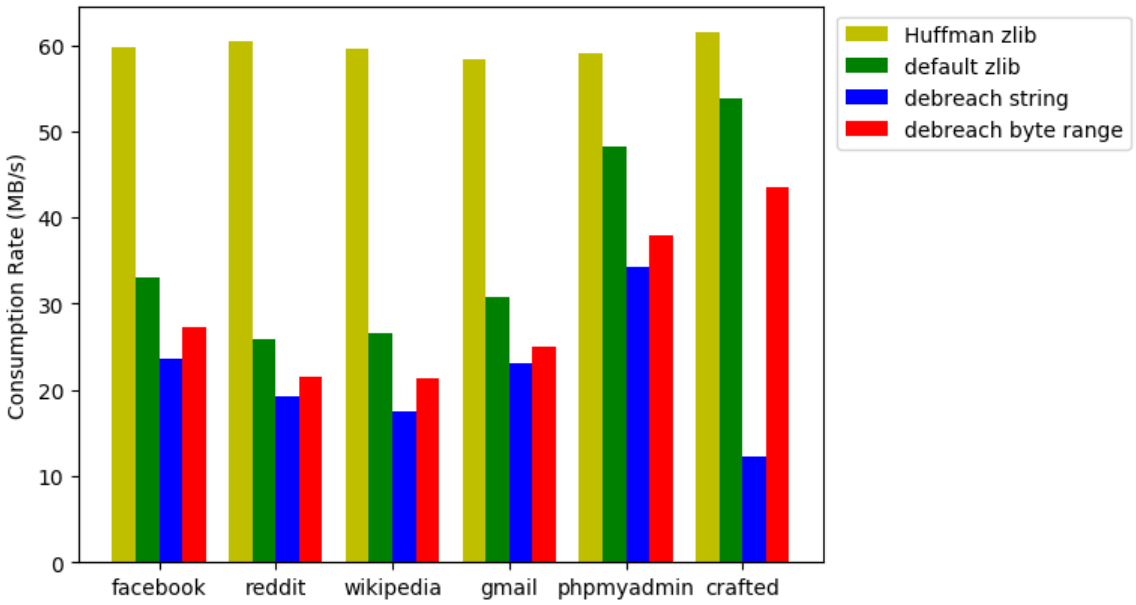

Figure 4.1: Consumption rates for each compression method on each data set. Each bar is the average of 60 trials.

Compared to standard zlib's consumption rates on the real data, string based matching is 6.61–13.84 MB/s (25.06–33.91%) slower, and the byte range based method is 4.33–10.36 MB/s (16.71–21.31%) slower. We note that, while a performance loss of 16–33% is significant, it means that debreach still maintains 84–67% of the performance of zlib while protecting token privacy. Byte range debreach performs better in all cases because no buffer searching is required. The difference in consumption rate between string and byte range debreach indicates that the buffer searching of string debreach incurs between 7% and 14% additional overhead because the two implementations are nearly identical in all other respects. As the number of partial matches of tainted strings increases, the consumption rate of string debreach degrades drastically, which is shown by its consumption rate on the crafted data. On the other hand, the byte range based method still performs well on this data.

site	default zlib	debreach string	debreach byte range
facebook	33.11	23.54	27.28
reddit	25.9	19.28	21.57
wikipedia	26.55	17.55	21.43
gmail	30.7	23.01	25.03
phpMyAdmin	48.15	34.31	37.89
crafted	53.87	12.35	43.5

site	debreach str delta	debreach br delta
facebook	-9.57 (28.92%)	-5.83 (17.61%)
rediit	-6.61 (25.54%)	-4.33 (16.71%)
wikipedia	-9.0 (33.91%)	-5.12 (19.3%)
gmail	-7.69 (25.06%)	-5.67 (18.47%)
phpMyAdmin	-13.84 (28.74%)	-10.26 (21.31%)
crafted	-41.52 (77.08%)	-10.36 (19.24%)

Table 4.3: Consumption rates for zlib, both debreach implementations, and the loss in consumption rate caused by debreach as compared to zlib. Rates are in MB/s.

Compared to the consumption rates of Huffman–only zlib on the real data, string based matching is between 25.0–42.0 MB/s (41–70%) slower, and the byte range based method is 21.0–39.0 MB/s (35–64%) slower. While this loss in consumption rate is substantial, debreach achieves far better compression ratios. Huffman–only also performs much better than standard zlib on all except the crafted data set. The small gain in consumption rate (7.6 MB/s) for Huffman–only zlib on the crafted data is surprising given that Huffman–only avoids many expensive operations of default zlib, such as keeping a dictionary lookup table and string comparisons.

site	Huffman zlib	string delta	byte range delta
facebook	59.75	-36.0 (60.61%)	-32.0 (54.34%)
reddit	60.45	-41.0 (68.1%)	-39.0 (64.31%)
wikipedia	59.5	-42.0 (70.51%)	-38.0 (63.99%)
gmail	58.36	-35.0 (60.57%)	-33.0 (57.11%)
phpMyAdmin	58.99	-25.0 (41.83%)	-21.0 (35.77%)
crafted	61.45	-49.0 (79.91%)	-18.0 (29.21%)

Table 4.4: Consumption rates for Huffman–only zlib and the difference in consumption rate caused by debreach as compared to Huffman–only zlib. Rates are in MB/s.

Effective Throughput

We plot the effective throughputs (as defined in equation 4.5) achieved for each compression method on each set of data in figure 4.2. For a given available bandwidth in a given graph, the algorithm with the top–most line is the optimal choice. Overall, debreach reduces the bandwidth at which it is the optimal choice as compared to default zlib by 24–48 Mb/s (17–20%) for byte range and 40–72 Mb/s (25–35%) for string. Interestingly, phpMyAdmin (29% loss) was not the largest loss for string debreach, rather wikipedia was (35% loss). This is likely due to the number of unique tokens in Wikipedia. Wikipedia has up to four unique tokens in each page, whereas phpMyAdmin has only one. For all sites, debreach is overtaken by Huffman coding. The available bandwidth where Huffman coding overtakes debreach varies between 96–184 Mb/s for string and 120–208 Mb/s for byte range. We note here that our test machines are relatively weak; faster machines would likely increase the consumption rates of all compression methods similarly, thereby increasing the available bandwidths where debreach is preferable.

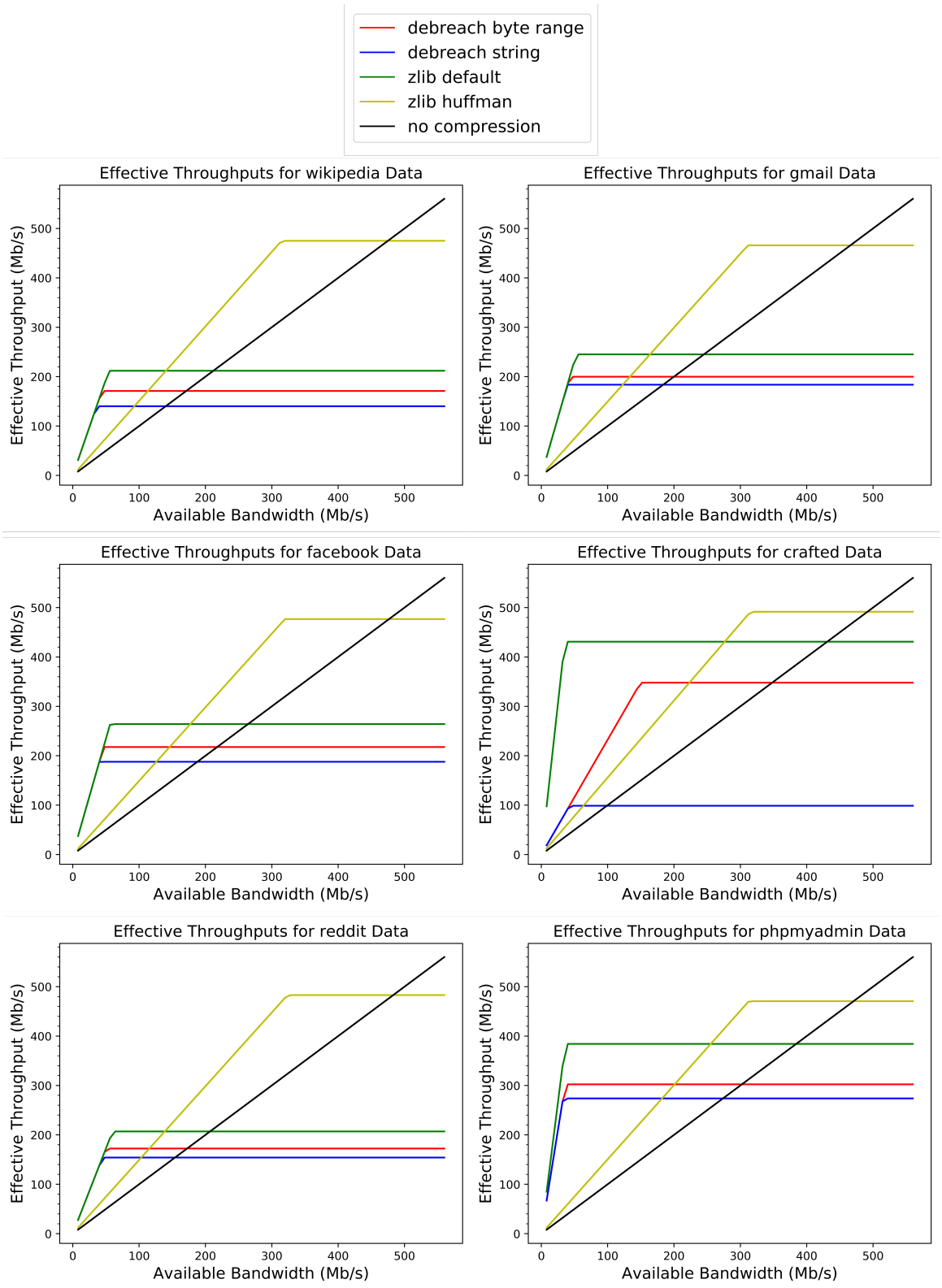

Figure 4.2: Graphs of the effective throughputs achieved on each site's data by each compression method for token tainting.

4.3.2 Arbitrary Tainting

Compression Ratio

We measure the impact on compression ratios that debreach has with varying amounts of taintedness in the input data. Like in the token tainting scenario, we only experiment on data that was compressed by the servers. We taint data by dividing each file into uniform sized chunks of size equal to 0.25% of the file's total size or 10 bytes, whichever is larger. We then randomly taint chunks of each file such that the tainted proportion is as close to the desired proportion as possible. The results further illustrate that the loss incurred by debreach depends not only on the proportion of tainted data but also the compressibility of the data. We measure the loss compared to zlib on each data set, tainting between 10 – 100% of the input in steps of 10%. The results are summarized in figure 4.3. We observe that the compression ratios of the least compressible data sets increases the slowest, and vice versa for the most compressible data sets.

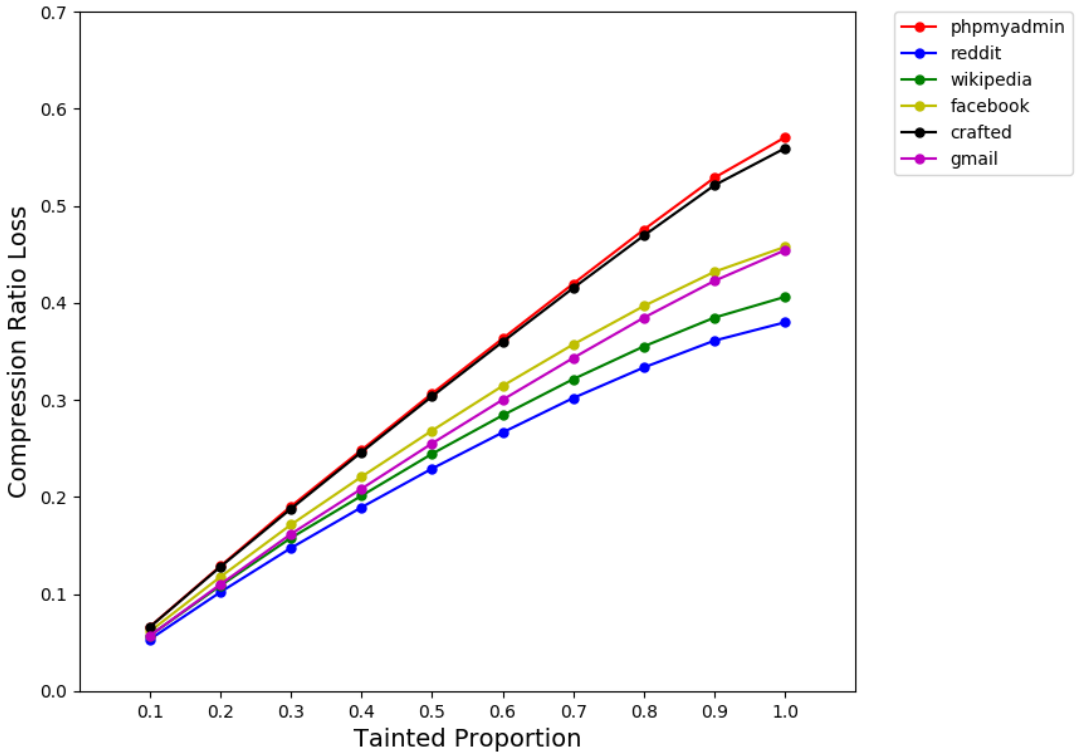

Figure 4.3: A graph capturing the impact on compression ratio as more input data is tainted. The x-axis is the proportion of input data that was tainted for the given data set. The y-axis is the compression ratio loss of debreach compared to zlib.

Consumption Rate

We measure the consumption rate in the same scenario. The results are graphed in figure 4.4. For all data sets except gmail, we see a small decline in the consumption rate initially, and then it increases steadily as the tainted proportion moves towards 100%. The initial loss of consumption rate is interesting because debreach performs more Huffman coding as the tainted proportion increases, and Huffman coding is a faster process than matching. We would expect the consumption rate to increase steadily. We hypothesize that this is because interrupting a match early is costly. When a match is stopped early due to intersecting a tainted region, debreach must

perform additional processing on the data that would have been matched, whereas standard zlib would not. Naturally, the lower proportions have more separate tainted regions, so matches are more likely to be interrupted. At 100% tainting, the consumption rates converge around 53 MB/s – a loss of around just 7 MB/s (11.7%) as compared to Huffman–only zlib.

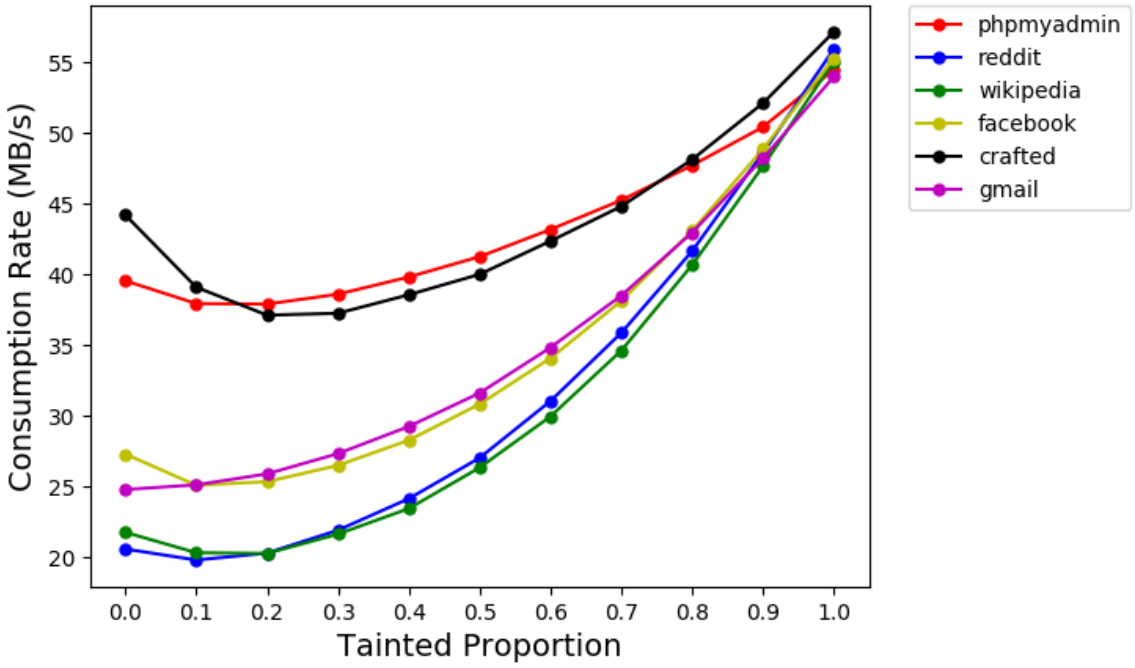

Figure 4.4: A graph showing the effect of increasing amounts of tainted data on the consumption rate for debreach.

Effective Throughput

We plot the maximum available bandwidth where debreach is the optimal choice versus Huffman–only and no compression for each site's data at each proportion of taintedness in figure 4.5. This measure is important because Huffman–coding and no compression are the only other completely secure, compressor–level solutions (except for SafeDeflate, but we cannot accurately measure the effective throughput of it for reasons discussed in section 4.3.4). The plot shows that the bandwidth

where debreach is optimal increase as the taintedness increases. At 10% taintedness, debreach is optimal for bandwidths between 112–208 Mb/s. At 50%, it is optimal between 144–224 Mb/s, and at 90% it is optimal between 256–272 Mb/s. At first, this may appear to indicate that increasing taintedness improves performance, however the *amount* of benefit achieved actually decreases as the taintedness increases, which the graph does not show.

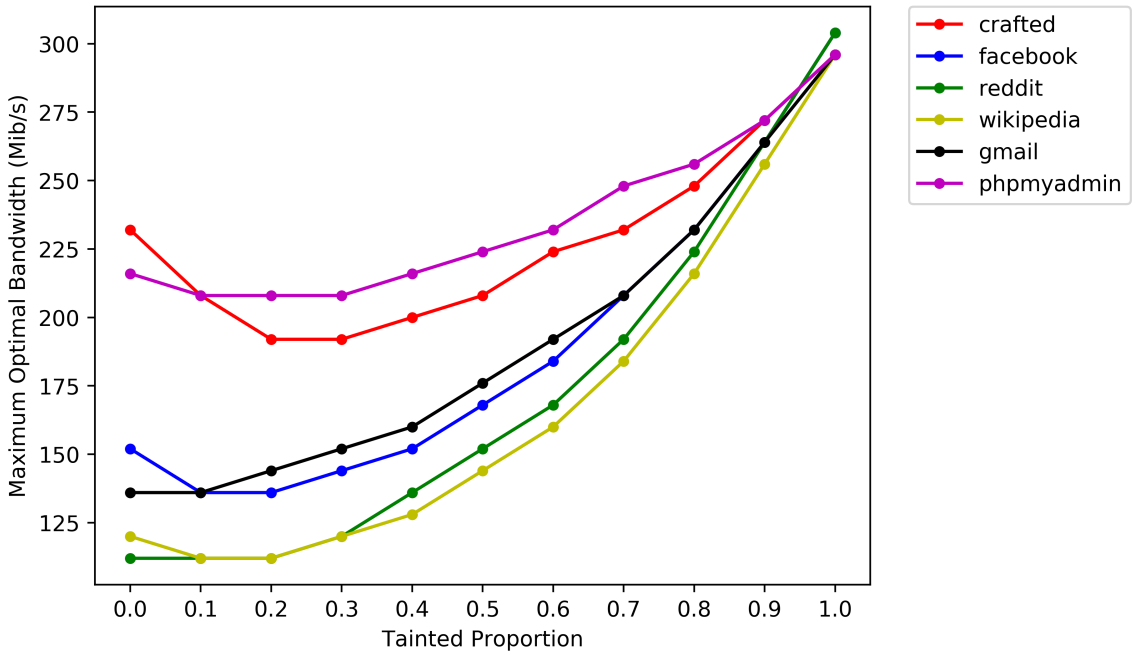

Figure 4.5: A graph showing the effect of increasing amounts of tainted data on the optimality of debreach. The y–axis shows the maximum available bandwidth where debreach outperforms Huffman coding. The x-axis is the proportion of data that is tainted.

Next, we show the effect that debreach has on effective throughput as compared to default zlib and no compression. We plot the difference in effective throughput between debreach and default zlib at each proportion of taintedness for the available bandwidths 8 Mb/s, 80 Mb/s, and 160 Mb/s in figures 4.6, 4.7, and 4.8. We limit the plots to these three available bandwidths for two reasons. First, providing a plot at all available bandwidths would require an excessive amount of space, and, second, many

of the plots would not lend additional insight into the effect of debreach as compared to zlib and no compression. In addition, the 8 Mb/s and 80 Mb/s bandwidths capture the minimum and maximum differences in effective throughput between debreach and zlib for each site, and the 160 Mb/s bandwidth captures the scenario where compression comes close to being unprofitable.

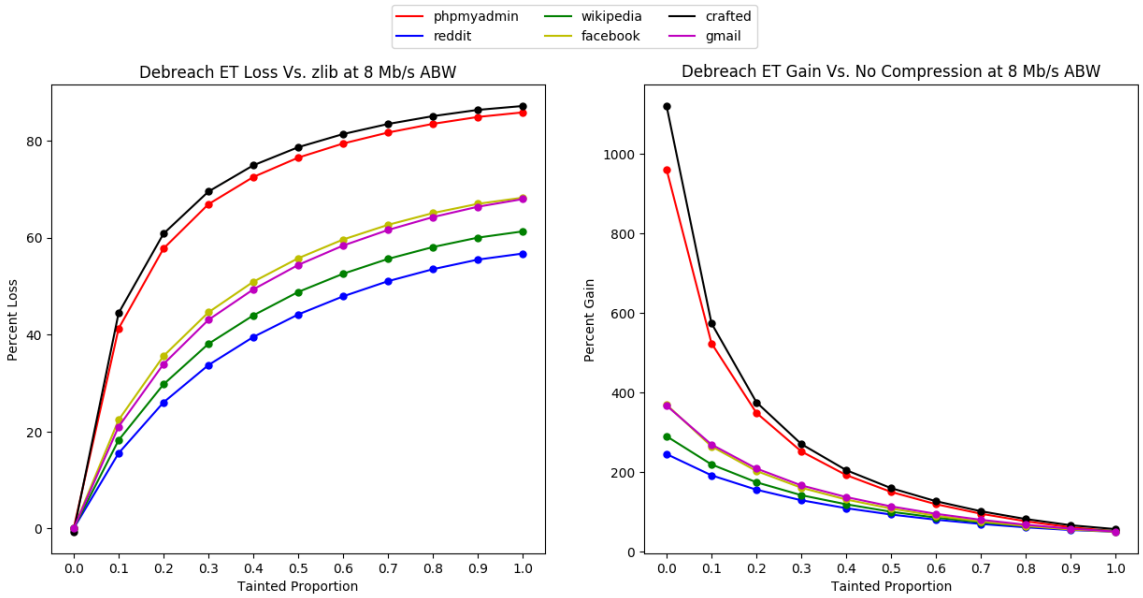

Figure 4.6: A graph showing the effect of increasing amounts of tainted data on the effective throughput of debreach compared to default zlib and no compression. The y–axis shows loss or gain calculated as the difference in effective throughput between debreach and the other method divided by the other method's effective throughput.

At 8 Mb/s bandwidth, debreach incurs the greatest loss as compared to default zlib on the most compressible data, namely phpMyAdmin and the crafted data. At 10% taintedness, debreach is between 41–43% slower than default zlib on this data. While the loss is substantial, debreach still improves the effective throughput by more than 500% over no compression. On the remaining data sets, debreach shows losses of just 16–22% at 10% taintedness with improvements between 191–269% over no compression. While the loss becomes much larger compared to zlib as the taintedness increases, overall debreach still performs well at this bandwidth. At 50% taintedness,

the improvement is between 93–149%, and at 100% taintedness the improvement is between 49–51%.

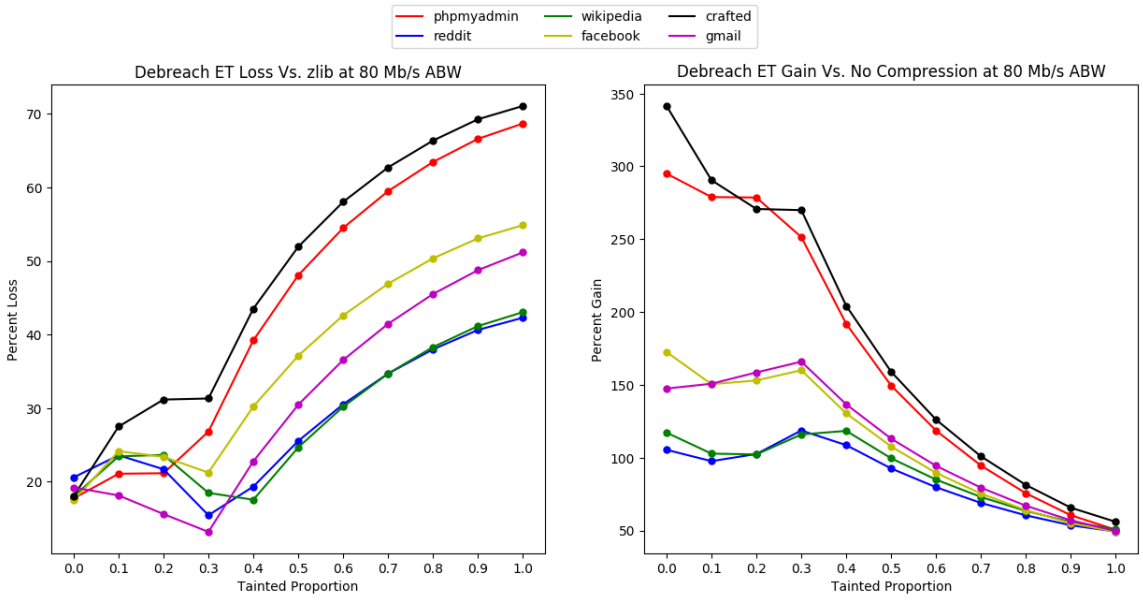

Figure 4.7: A graph showing the effect of increasing amounts of tainted data on the effective throughput of debreach compared to default zlib and no compression. The y–axis shows loss or gain calculated as the difference in effective throughput between debreach and the other method divided by the other method's effective throughput.

At 80 Mb/s bandwidth, the maximum loss in effective throughput across all real data sets decreases overall, and in particular it decreases from 41% to 24% at 10% taintedness. The improvement over no compression also decreases, though debreach still performs well in this respect. At 10% taintedness, it improves effective throughput by 295% on phpMyAdmin and between 106–175% on the other data data sets. For higher levels of taintedness, debreach still makes at least a 50% improvement over no compression.

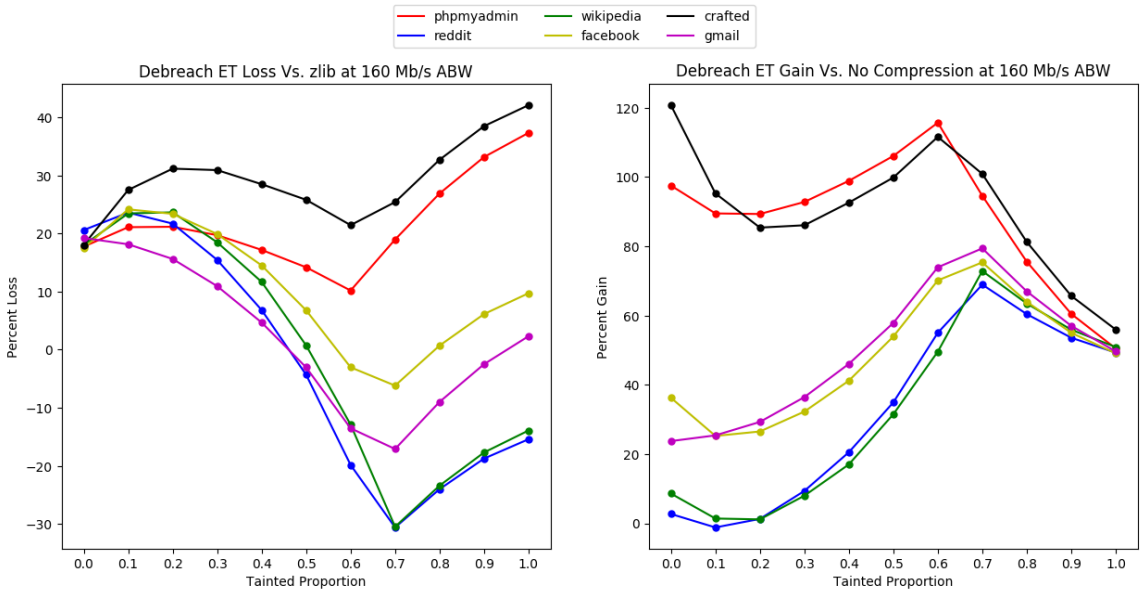

Figure 4.8: A graph showing the effect of increasing amounts of tainted data on the optimality of debreach. The y–axis shows the maximum available bandwidth where debreach outperforms Huffman coding. The x-axis is the proportion of data that is tainted.

At 160 Mb/s bandwidth, the maximum loss in effective throughput remains the same as in 80 Mb/s for lower levels of taintedness. In addition, debreach begins to overtake the effective throughputs of zlib at higher levels of taintedness because, while debreach outputs less compressed data, it outputs faster than zlib, so it is able to better saturate the available bandwidth. Compared to no compression, debreach still substantially improves effective throughput on the most compressible data (60–120%), though on the other data sets, it comes close to being a bottleneck.

4.3.3 CTX

Comparing debreach to CTX is difficult because CTX can only be used on content where JavaScript is allowed to execute, and it cannot be reliably applied to data that holds syntactical significance, such as HTML tags, CSS, and JavaScript. Given that security tokens are often embedded in HTML attributes, JavaScript, and JSON data, CTX is not fit for the task of protecting them, so we omit a comparison to debreach in

this scenario. We can however modify the second scenario of the previous sections to fit the capabilities of CTX by inserting secrets into the HTML content of our data sets (in a CTX–compatible way) and protecting them. We then compare the compression ratios achieved when protecting the data. Ideally, we would protect the real content in the HTML, but many of the files do not have enough CTX–protectable content to experiment on. As for consumption rates, we cannot make a fair comparison because CTX is implemented as a web framework module in python, whereas such a module does not yet exist for debreach.

We compare the compression ratios achieved by debreach and CTX on data with varying amounts of secrets and, in the case of CTX, origins. Secrets of an arbitrary length, 25 bytes in our case, are randomly picked from a movie transcript and inserted into each HTML document until the desired proportion of secrets is achieved. Since the protectable portion of HTML data is usually small, we experiment with only small proportions of secrets, specifically between 1% and 10% in increments of 1%. In addition, the number of origins can greatly affect the compression ratio achieved by CTX, so we experiment with a variety of numbers, specifically 1, 10, 25, and 50. The difference in compression ratios achieved on each proportion of secrets and number of origins are graphed in figure 4.9. Points below the x–axis indicate debreach performed better, and vice versa for CTX.

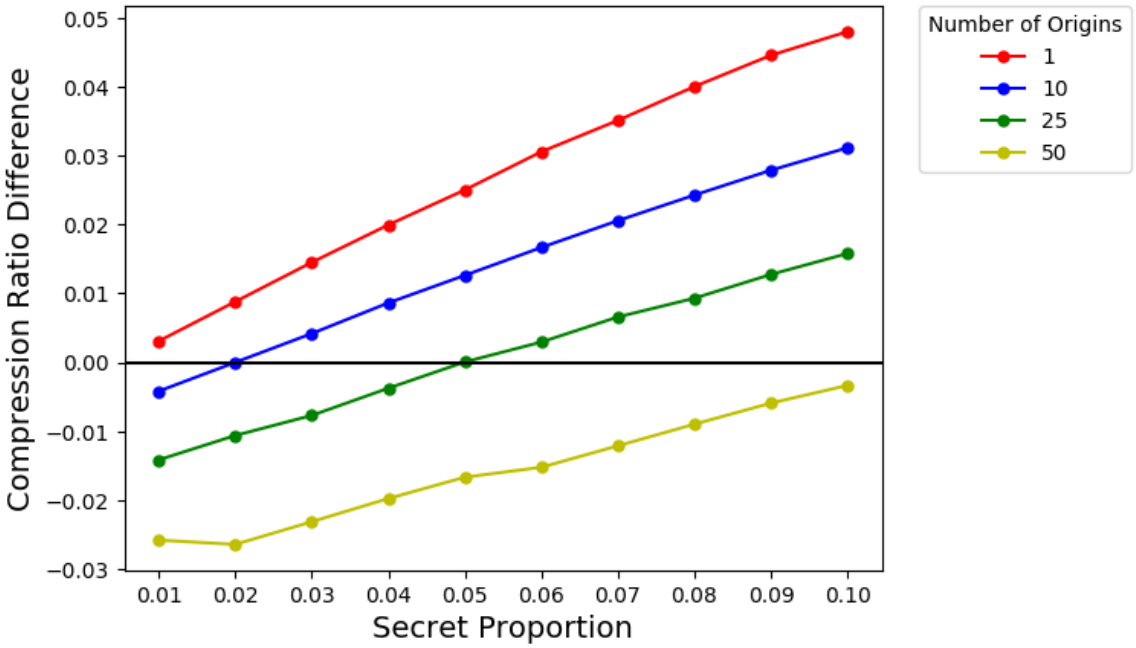

Figure 4.9: A graph of the differences in compression ratios achieved by CTX and debreach on varying proportions of secret data and origins.

For small amounts of secret data, debreach tends to perform better because CTX has a constant size overhead caused by the JavaScript for undoing the masking that must be included with the response. The size of the instructions are the same regardless of the amount of secret data. As the amount of secret data increases, CTX eventually overtakes debreach because it can cross-compress secrets from the same origin whereas debreach cannot. The point at which CTX overtakes debreach depends on the number of origins because fewer origins means cross-compression is more likely to occur.

The amount that the compression ratios differ by is small in all cases – less than 0.05 at most. The best case for CTX as compared to debreach is 10% secret data and one origin because all secrets can be compressed with each other whereas debreach Huffman codes them all. The worst case for CTX is 1% secret data and 50 origins because of the constant overhead and inability to cross compress.

4.3.4 SafeDeflate

We compare debreach to SafeDeflate in the token tainting scenario under the assumption that security tokens are drawn from a hexadecimal alphabet for all sites. That is, we configure SafeDeflate with the secret alphabet as hexadecimal characters. We give SafeDeflate this benefit for two reasons. First, this is the best case for SafeDeflate as reported by its author, so this ensures a best–case comparison. Second, configuring a website to use a different alphabet for security tokens would likely be easy. In addition, even though some of our test data has alphanumeric security tokens, changing a website's security token alphabet would cause little difference in the site's overall content because the tokens account for a very small proportion of the total data. Finally, to avoid putting debreach at an unfair disadvantage, we create a new set of worst case data with tokens drawn from a hexadecimal alphabet.

While a fair comparison of compression ratios can be made, a fair comparison of consumption rates cannot. This is because (in our experiments) SafeDeflate achieved consumption rates of around 7 MB/s on all data, which is unreasonably low for the algorithm described by its author. We believe the slowness is caused by two factors. First, SafeDeflate is currently implemented in the go language, which is inherently slower than C. Second, it is implemented as as a proof–of–concept and lacks many of the implementation optimizations present in zlib and debreach. If SafeDeflate was implemented in C and had the optimizations present in zlib and debreach, it would certainly perform significantly better. The compression ratio results are summarized in figure 4.10, and the numerical values are shown in table 4.5.

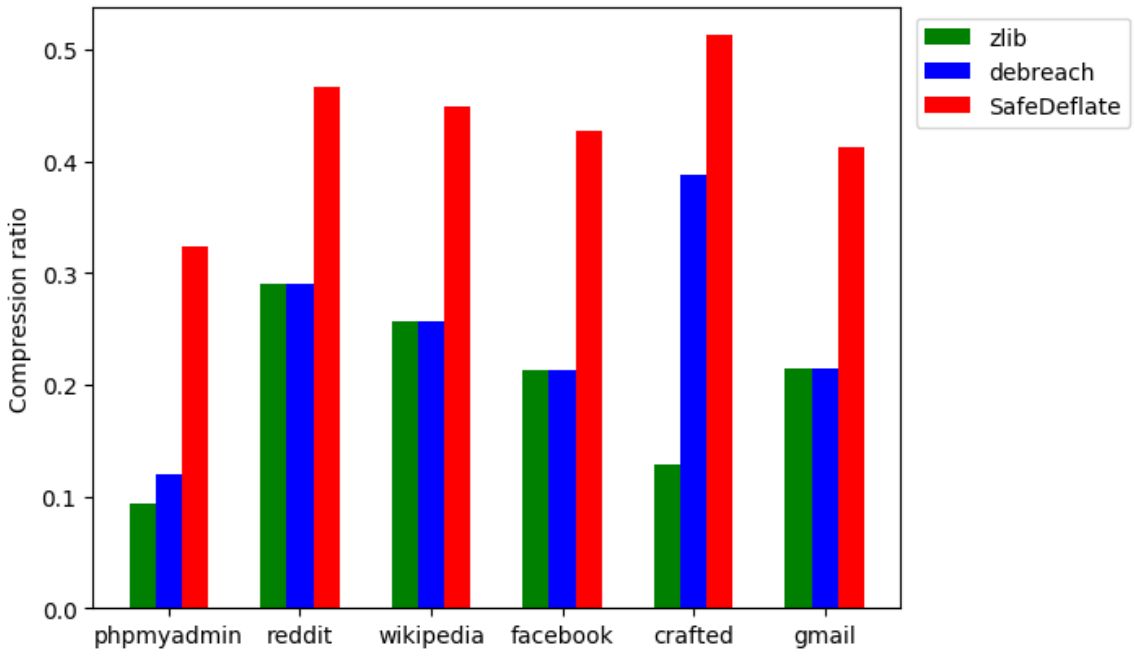

Figure 4.10: A graph of the differences in compression ratios achieved by CTX and debreach on varying proportions of secret data and origins.

site name	SafeDeflate	zlib	debreach
phpMyAdmin	0.26617	0.09424	0.11948
reddit	0.34706	0.29006	0.29016
wikipedia	0.31602	0.25666	0.25684
facebook	0.29813	0.21317	0.21332
crafted	0.48241	0.12819	0.38808
gmail	0.32395	0.21426	0.21433

Table 4.5: Compression ratios achieved by SafeDeflate with a hexadecimal secret alphabet as compared to zlib and debreach.

On real data, SafeDeflate loses between 5–17% as compared to standard zlib, which is significant but does not make SafeDeflate impractical. Debreach performs between 5–14% better across all datasets, which we would expect because SafeDeflate avoids many safe matches because they "appear" unsafe. For token tainting, the compression ratios of SafeDeflate are still good – a reduction to 27–35% of the original size would still yield benefit in practice, but is far from optimal.

For completeness, we include the results of SafeDeflate with an alphanumeric alphabet in table 4.6. For four out of five websites, the compression ratios are near or greater than double zlib's with overall losses ranging from 17–22%.

site name	SafeDeflate	zlib	debreach
phpMyAdmin	0.32339	0.09424	0.11948
reddit	0.46647	0.29006	0.29016
wikipedia	0.44931	0.25666	0.25684
facebook	0.42677	0.21317	0.21332
crafted	0.51296	0.12819	0.38808
gmail	0.41221	0.21426	0.21433

Table 4.6: Compression ratios achieved by SafeDeflate with an alphanumeric secret alphabet as compared to zlib and debreach.

5 Conclusions

5.1 Security of debreach

We acknowledge that string based debreach is vulnerable to brute-force, $O(x^n)$ (exponential) complexity attacks where n is the length of the secret string and x is the size of the alphabet from which the string is drawn. The brute–force attack could be carried out as follows. The attacker injects their guess at a target secret twice into the document containing the target secret. If the two identical guesses do not match the target secret, the guesses will compress with themselves. If the guesses do match the target secret, they will not compress at all, and the attacker will see a sudden increase in the compressed size. The attacker keeps performing this process with a new guess until he sees the sudden increase in size, which on average will take $x^n \div 2$ guesses. An $O(x^n)$ attack is acceptable for security purposes, however the complexity depends on the random nature of the string being protected. We acknowledge that the complexity of the aforementioned attack can be reduced to practical ranges if the unsafe string is not randomly generated. For this reason, the string based algorithm should only be used for randomly generated data. This is not an issue for security tokens because they should always be randomly generated, but it may be an issue for other data such as email addresses. Byte range based debreach fixes the performance and security shortcomings of string based debreach, though it shifts more burden on the developer.

5.2 Comparison to other Approaches

5.2.1 Huffman Coding and No Compression

The effective throughputs achieved by each method clearly show when debreach is preferable to both Huffman coding and no compression. For token tainting, debreach is preferable at bandwidths of at least up to 96 Mb/s for string based and 120 Mb/s for byte range based, which certainly exist in the real world. For arbitrary tainting, we've shown that debreach is preferable at bandwidths up to at least up to 112 Mb/s depending on the amount of taintedness. Thus, debreach would often be preferable to Huffman coding if it was applied correctly.

5.2.2 CTX

Debreach addresses the shortcomings of CTX, namely the ability to protect arbitrary content of all data types. Given the prevalence of content sent in data formats other than HTML, we believe this reason alone is enough to favor debreach over CTX. Furthermore, while CTX has the advantage of securely allowing cross-compression, the advantage is minimal; generally debreach performs equally or slightly better in cases where CTX can be applied. And while a direct comparison of effective throughputs is not currently possible, the bulk of the overhead that would be caused by using debreach would likely be in the compressor itself. We have shown that byte range based debreach performs comparably to zlib, which strongly suggests that it would be practical for real–world use. Overall, there appear to be limited advantages and significant disadvantages of CTX over debreach.

5.2.3 SafeDeflate

SafeDeflate is vulnerable to the same attack previously discussed in section 5.1, so it should only be used for randomly generated secrets. Whether SafeDeflate or debreach should be used depends on the use case. SafeDeflate has the advantage of being easier to use than debreach, so it is preferable when fast integration is top priority. SafeDeflate comes with a cost though; in its best case where secrets are drawn from a hexadecimal alphabet, SafeDeflate produces compressed sizes that are 20–155% larger than debreach. With this in mind, debreach is preferable when time can be spent on integration, and performance is important.

5.2.4 Token Masking Frameworks

For token tainting, token masking frameworks have a performance advantage over compressor–level approaches. First, the primary operations of token masking, namely generating one–time pads and applying XOR, are fast, so they are unlikely to affect the compressed output rate of the compressor. They also have little impact on compressibility because the tokens and pads can securely compress with each other. With these two points in mind, a token masking framework would likely have an insignificant effect on the effective throughput achieved in practice, while we've already shown that debreach has an impact on effective throughput. SafeDeflate would likely see an impact on effective throughput as well given its compression ratio loss. Furthermore, token masking frameworks are completely secure and easy to use when implemented properly. For example, to use Django–debreach, the developer simply enables a module installed through a package manager. Overall, token masking frameworks appear preferable for security token privacy, through we see two potential cases where debreach may be preferable.

The first area where a token masking framework may present an issue is in integration with an existing application. If an application implements its own security token mechanisms, then migrating to a token masking framework may entail reworking a substantial amount of source code. For example, phpMyAdmin directly handles security tokens throughout its codebase without function calls, so each point in the code where the token is handled would need to be updated to integrate a token masking framework. While this certainly would not be impractical, we've shown how debreach can be integrated in phpMyAdmin with just two additional statements in the codebase.

In addition, token masking frameworks work well in web applications because they are implemented on the server–side, and only the client does not handle the token directly. Thus, there could not possibly be compatibility issues. However, masking may be more difficult to apply in scenarios where both ends may need to know how to handle the masked data. In cases where both ends need to handle the token, debreach would arguably be better suited. Of course, such a case is purely theoretical at this time.

5.3 Future Work

5.3.1 Efficiency Improvements

During development, two potential optimizations became apparent. First, the additional memory usage of both string and byte range debreach can be cut in half by reducing the maximum allowed match from 258 to 254 bytes. This reduction would allow the *next_taint* metadata array to store 8–bit integers rather than 16–bit integers. This may or may not increase the consumption rate as well.

Second, while implementing string debreach, we operated under the assumption

that whole tokens needed to be excluded from dictionary compression to guarantee security. The author of SafeDeflate has shown this is not true though; matches only cannot begin or end inside a token, but whole tokens can be securely matched with other whole tokens. String debreach could be reworked to allow whole tokens to match with each other thereby reclaiming all lost compression ratio, and possibly some of the consumption rate loss. To implement this logic, one would simply need to ensure that a match neither begins nor ends in a whole token. For string debreach, whole tokens correspond to the tainted regions.

5.3.2 Framework

Currently, the biggest area of concern for BREACH–like attacks is web applications. Since developers do not generally write their applications in C, a framework needs to be made for debreach to be useful. A token protecting framework would be practical in terms of usability, security, and efficiency, but it would likely perform equal to or worse than a token masking framework as was discussed in 5.2.4. A general purpose framework for string debreach could be made to give developers direct access to the string–tainting functionality, but we suspect this may lead to false assumptions of security — A developer may overlook the fact that non–randomly generated data cannot be protected with string tainting. On the other hand, a general purpose protection framework similar to CTX could be made for byte range debreach. We envision implementing a function that outputs protected data to the client. For example, the CTX framework implements a function *protect()* that handles data which should be protected. A debreach framework could have a similar function to handle data that should be excluded from dictionary compression.

5.3.3 Multiple Dictionaries

The main advantage of CTX over debreach is allowing cross–compression between data marked with the same origin. Currently, debreach does not support marking data with an origin; data is separated into two "origins", safe and unsafe, where unsafe data is excluded from dictionary compression. At the expense of some additional processing or memory, data could be marked with an origin, and dictionary matching could be allowed between data with the same origin. Such a mechanism would close the compression ratio performance gap between debreach and CTX; in fact, debreach would likely perform better because the *div* tags and additional JavaScript inserted by CTX would be unnecessary in debreach.

Marking data with an origin would require storing additional metadata about the window. One way to do this would be storing a metadata array similar to the *next_taint* array that stores the origin. The *next_taint* array then stores the distance to the next origin boundary. This way, the maximum length for a match beginning at index i is *next_taint[i]*.

Next, we need to be able to quickly find potential matches. One way to do this is to store a canonical dictionary for each origin. As the deflate window is processed, the dictionary corresponding to a data's origin is updated. The memory for this approach would be somewhat large. The standard deflate dictionary is 256^3 bytes = 16KB in size, so each additional origin would add another 16KB to the memory requirements. This approach has the advantage of being able to quickly find potential matches for strings of a particular origin.

Alternatively, all strings could be stored in the same dictionary, and the compressor would check that the origins match after querying the dictionary. This approach significantly reduces the memory overhead, especially for large amounts of origins, but slows down the process of finding good matches.

5.3.4 Integration with Taint Tracking

Some common web application development frameworks include taint tracking. For example, ruby, perl, python, and PHP all have taint tracking either built–in or as an extension. Debreach could potentially be integrated with any of these language's tainting mechanisms to passively not compress tainted input. Unfortunately, developers are generally forced to untaint data before it is used in a meaningful way, which would prevent the taint information from reaching debreach. Another bit of information indicating whether data was tainted at any time would need to be tracked in order for debreach to be used properly in this scenario. Still, this integration would provide passive protection from compression side–channels, so it is worth noting.

5.4 Final Remarks

We've shown that selected portions of data can be efficiently excluded from dictionary compression for the purpose of ensuring privacy of the selected data. Data can be selected with either string matching or byte range marking. While token masking appears preferable for the specific case of protecting security tokens, compressor–level approaches to ensuring privacy have a significant advantage in the realm of generality. Practical next steps include making a framework for byte range debreach and experimenting with a multiple dictionary implementation of debreach.

Bibliography

[1] R. Barnes, M. Thomson, A. Pironti, and A. Langley. *Deprecating Secure Sockets Layer Version 3.0*. RFC 7568. RFC Editor, 2015. URL: https://www.rfc-editor.org/rfc/rfc7568.txt (cit. on p. 4).

[2] W. Chang, B. Streiff, and C. Lin. "Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: ACM, 2008, pp. 39–50. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455778. URL: http://doi.acm.org/10.1145/1455770.1455778 (cit. on pp. 74, 79).

[3] E. Chin and D. Wagner. "Efficient Character-level Taint Tracking for Java". In: *Proceedings of the 2009 ACM Workshop on Secure Web Services*. SWS '09. Chicago, Illinois, USA: ACM, 2009, pp. 3–12. ISBN: 978-1-60558-789-9. DOI: 10.1145/1655121.1655125. URL: http://doi.acm.org/10.1145/1655121.1655125 (cit. on p. 75).

[4] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. RFC Editor, 2008. URL: https://www.rfc-editor.org/rfc/rfc5246.txt (cit. on p. 3).

[5] T. Duong and J. Rizzo. "The CRIME Attack". ekoparty. 2012. URL: https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit (cit. on pp. 1, 12).

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. RFC Editor, 1999. URL: https://www.rfc-editor.org/rfc/rfc2616.txt (cit. on p. 3).

[7] J.-l. Gailly and M. Adler. *zlib*. Version 1.2.8. URL: https://zlib.net/ (cit. on p. 36).

[8] Y. Gluck, N. Harris, and A. Prado. "BREACH: Reviving the CRIME Attack". Black Hat USA 2013 Conference Paper (cit. on pp. 1, 11, 13).

[9] Y. Gluck, N. Harris, and A. Prado. "SSL, Gone in 60 Seconds: A BREACH Beyond CRIME". Black Hat USA 2013 Conference Presentation. July 2013 (cit. on pp. 11, 12).

[10] D. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". Undetermined. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898 (cit. on p. 7).

[11] D. Karakostas, A. Kiayias, E. Sarafianou, and D. Zindros. "CTX: Eliminating BREACH with Context Hiding". In: *Black Hat EU 2016*. Nov. 2016. URL: https://www.blackhat.com/docs/eu-16/materials/eu-16-Karakostas-Ctx-Eliminating-BREACH-With-Context-Hiding-wp.pdf (cit. on p. 16).

[12] D. Karakostas and D. Zindros. "Practical New Developments on BREACH". In: *Black Hat Asia 2016*. Mar. 2016. URL: https://www.blackhat.com/docs/asia-16/materials/asia-16-Karakostas-Practical-New-Developments-In-The-BREACH-Attack-wp.pdf (cit. on pp. 1, 14).

[13] J. Kelsey. "Compression and Information Leakage of Plaintext". English. In: *Fast Software Encryption*. Ed. by J. Daemen and V. Rijmen. Vol. 2365. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 263–276. ISBN: 978-3-540-44009-3. DOI: 10.1007/3-540-45661-9_21. URL: http://dx.doi.org/10.1007/3-540-45661-9_21 (cit. on pp. 10, 11).

[14] B. Livshits. "Dynamic taint tracking in managed runtimes". In: *Technical Report MSR-TR-2012-114, Microsoft* (2012) (cit. on p. 15).

[15] A. Naderi-Afooshteh, A. Nguyen-Tuong, M. Bagheri-Marzijarani, J. D. Hiser, and J. W. Davidson. "Joza: Hybrid Taint Inference for Defeating Web Application SQL Injection Attacks". In: *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE. 2015, pp. 172–183 (cit. on p. 75).

[16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. "Automatically Hardening Web Applications Using Precise Tainting". English. In: *Security and Privacy in the Age of Ubiquitous Computing*. Ed. by R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura. Vol. 181. IFIP Advances in Information and Communication Technology. Springer US, 2005, pp. 295–307. ISBN: 978-0-387-25658-0. DOI: 10.1007/0-387-25660-1_20. URL: https://scholar.google.com/citations?view_op=view_citation&hl=en&user=4QDvnc4AAAAJ&citation_for_view=4QDvnc4AAAAJ:u5HHmVD_uO8C (cit. on pp. 74, 77).

[17] R. Peon and H. Ruellan. *HPACK: Header Compression for HTTP/2*. RFC 7541. RFC Editor, 2015. URL: http://www.rfc-editor.org/rfc/rfc7541.txt (cit. on p. 2).

[18] J. Postel. *Internet Protocol*. RFC 791. RFC Editor, 1981. URL: https://www.rfc-editor.org/rfc/rfc791.txt (cit. on p. 5).

[19] J. Postel. *Transmission Control Protocol*. RFC 793. RFC Editor, 1981. URL: https://www.rfc-editor.org/rfc/rfc793.txt (cit. on p. 5).

[20] J. Salowey. *Confirmation of Consensus on Removing Compression from TLS 1.3*. Mar. 2014. URL: https://www.ietf.org/mail-archive/web/tls/current/msg11619.html (cit. on p. 2).

[21] M. Vanhoef and T. V. Goethem. "HEIST: HTTP Encrypted Information can be Stolen through TCP-windows". In: *Black Hat USA 2016*. July 2016. URL: https://www.blackhat.com/docs/us-16/materials/us-16-VanGoethem-HEIST-HTTP-Encrypted-Information-Can-Be-Stolen-Through-TCP-Windows-wp.pdf (cit. on p. 1).

[22] W3Techs. *Usage of Gzip Compression for websites*. 2017. URL: https://w3techs.com/technologies/details/ce-gzipcompression/all/all (cit. on p. 2).

[23] X. Zheng, J. Jiang, J. Liang, H.-X. Duan, S. Chen, T. Wan, and N. Weaver. "Cookies Lack Integrity: Real-World Implications." In: *USENIX Security*. 2015, pp. 707–721 (cit. on p. 29).

[24] M. Zieliński. *SafeDeflate: compression without leaking secrets*. Cryptology ePrint Archive, Report 2016/958. 2016. URL: https://eprint.iacr.org/2016/958.pdf (cit. on p. 18).

[25] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *Information Theory, IEEE Transactions on* 23.3 (May 1977), pp. 337–343. ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714 (cit. on p. 8).

A Appendix A

A.1 BREACH Implementation

Our experimental BREACH environment uses three Ubuntu 14.04 boxes. Their physical connections are shown in figure A.1. Our eavesdropper, Eve, has three physical ethernet ports, and the victim and the mail server each have one. Eve is physically interposed between the victim and mail server, and she runs a bridge so that she can transparently pass traffic through herself. The bridge essentially turns Eve into a router.

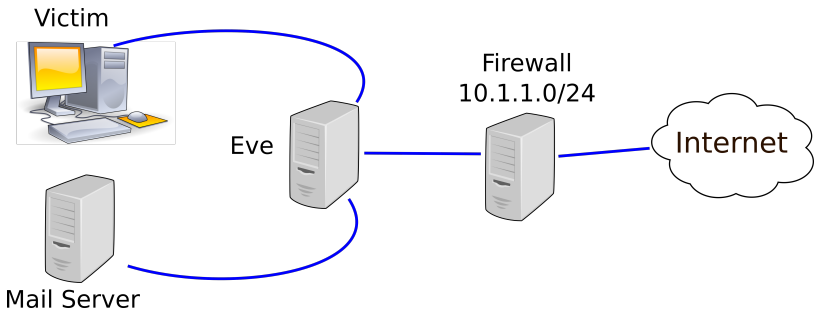

Figure A.1: A physical view of the experimental setup

The mail server uses postfix and dovecot to handle mail protocols, and it runs squirrelmail on top of a typical LAMP setup for web mail. It runs Apache 2.2.0 configured to only accept HTTPS connections and PHP 5.0.1. Our victim accesses the web mail service through a Chromium web browser. Eve runs a simple Java HTTPS servlet which acts as a BREACH oracle. We explain the BREACH oracle next.

A.2 BREACH Oracle

Figure A.2 shows half of a round of the "two guesses" method which is explained in section 2.1.3. The BREACH oracle can be broken into two parts: a piece of JavaScript which is injected into the victim's browser, and the oracle itself, which is a Java HTTPS server that runs on the assailant's machine. We explain the JavaScript part first.

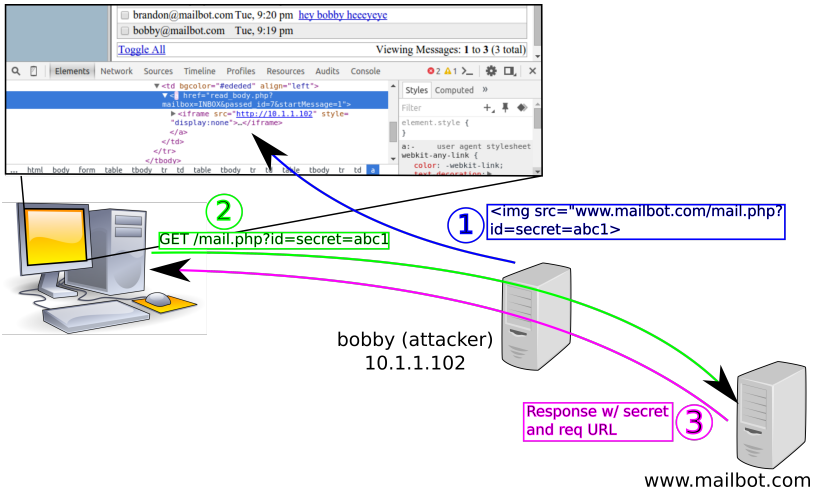

Figure A.2: A scenario in which a BREACH attack occurs. Our attacker, bobby, sends the victim an email with an iframe in the subject line, and the squirrelmail web client fails to sanitize it. Our victim then accesses the webmail client, and the attack begins. This figure depicts half of a round of the "two guesses" method.

For our experiments, we assume the attacker can inject a piece of JavaScript into the victim's browser. In our case, we inject an iframe into the victim's browser, and the iframe requests the JavaScript from the attacker. Figure A.2 shows that bobby has managed to inject an iframe in the subject line of the victim's webmail client. The purpose of the JavaScript is create partially chosen plaintexts on behalf of the oracle. This is shown in steps 1 and 2 of figure A.2. In our implementation, the JavaScript makes a GET request to the oracle, and the oracle responds at its convenience with the guess it wants to try next. Every time the JavaScript receives a response, it crafts a request containing the guess and sends it to the mail server. Then it requests the next guess from the oracle in a callback function, and the cycle repeats.

The BREACH oracle is implemented as a Java HTTPS servlet that runs on the attacker's machine. It simply waits for requests from the JavaScript, and then responds based on internal state variables. The basic logic can be deduced from the

request handle

```
public void handle(HttpExchange exch) throws IOException {
  OutputStream response = exch.getResponseBody();
  Headers headers = exch.getResponseHeaders();
  headers.add("Content-Type", "text/plain; charset=iso-8859-1");
  if (aligning) {
    doAlignment(exch, response, headers);
  } else if (!secondGuess) {
    doFirstGuess(exch, response, headers);
  } else {
    doSecondGuess(exch, response, headers);
    isGuessCorrect();
  }
}
```

Note that the BREACH oracle is single–threaded, so it will handle requests one at a time in the order they were received. This implementation of a BREACH oracle suggests three phases: aligning, first guess, and second guess. In each of these phases, the oracle responds to the request with the string it wishes to inject into the mail server's response. After we respond, the handler sleeps, and we pass control to a packet sniffer which measures the server's response.

In the alignment phase, the oracle attempts to determine the precise amount of padding needed to cause the response to roll over into a new cipher block. Once this amount is found, we subtract one character from the padding, so that when we try an incorrect guess, the response rolls over to a new cipher block, but when we try an incorrect guess the response will not roll over. Once we've aligned the response to this tipping point we proceed to the first guess phase. Note that this step is not

necessary when a stream cipher is used.

What happens in the first and second guess phase is mostly explained in section 2.1.3. We also include some handling when something clearly went wrong. For example, when the measured response size is much larger or smaller than the responses previously, we try the guess again. In addition, if the second guess didn't cause the response to tip, then we try changing the padding character.

We note that our BREACH oracle could be improved in many ways, however the purpose of the BREACH oracle is to show we can prevent BREACH. If we can show user input is never compressed, then we can reasonably claim BREACH will be thwarted.

A.3 Taint Tracking

A.3.1 Classification of Taint Tracking Systems

Livshits defines four characteristics for classification: level of instrumentation, tracking granularity, form of tainting, and the type of flow tracked. Each characteristic is discussed in turn.

Level of Instrumentation. Implementing a taint tracking system always entails writing code to enforce the policy. The level of instrumentation refers to where this code lives and how it acts on the application. Livshits describes five levels, but we omit discussions of those irrelevant to this work. The highest level is source–instrumentation. In this approach, the source code is directly modified to implement the policy code, so no special infrastructure is required. An issue with this approach is that implementing a policy at the source level can be cumbersome for large applications because every source, sink, and propagator in the application's source code must be instrumented. In addition, the implementation is specific to the application; usu-

ally it cannot be easily ported to another application. Chang, et al. present a source level taint tracking system which addresses this issue by automating instrumentation of the source code through a modified C compiler [2]. The other frequently appearing level of instrumentation is at the library level. This level of instrumentation achieves portability by modifying the language and functions used by the application rather than the application itself. This also means that policy enforcement is inherent in the language which typically means the application needs less instrumentation (or none at all). For example, Anh Nguyen-Tuong, et al. present an augmented PHP interpreter which can passively prevent cross–site scripting attacks by automatically sanitizing any output derived from tainted data [16]. Several other library level instrumentations have been proposed as well, and they are discussed in section A.3.2.

Tracking Granularity. Tracking granularity refers to the precision with which the taint tracking system records taintedness. The common granularities are byte, character, string, and object. The strategy we take for preventing compression vulnerabilities is to not compress user input. To us, the usefulness of taint tracking comes from knowing what data are derived from user input. Thus string granularity tracking is not optimal with our approach because HTML pages are often built into a single string and then sent back to the client, so it may be the case that all data would not be compressed. A similar problem can occur with object granularity. In addition, tracking at the primitive level (byte, int, long, etc.) lacks a straight–forward approach. In object–oriented languages like Java, taint information can be stored directly with primitives if they are wrapped in an object. For example, a string object in Java can be augmented with an array that contains the taint information on each individual primitive. However, taint information cannot be stored directly with primitives in all languages (without special hardware support), so instead it must be stored in a separate data structure. Chang, et al. present an interesting run–time

library for C that accomplishes this.

Form of Tainting. Taint tracking systems can be used in two complementary ways which amount to black–listing and white–listing. In the black–list approach, also known as "negative tainting", we mark data as untrusted and everything else is considered trusted. In the white–list approach, also known as "positive tainting", we mark data as trusted and everything else is considered untrusted. Generally, trusted data is far more abundant than untrusted data, so negative tainting is usually preferred because of the lower performance overhead. However, positive tainting has proven practical in an area similar to taint tracking known as taint inference. In addition, both types of tainting can be used in tandem [15].

Type of flow. Taint tracking systems can track both explicit and, rarely, implicit flow. Explicit flow is when data is derived directly from other data members, for example, when two strings are concatenated. Implicit flow usually refers to when data is derived as the result of a truth value. For example, in the statement: if A then B = C else B = D, data has implicitly flowed from variable A to variable B because the value of A determined the value of B. We are only concerned with data derived directly from user input, so we ignore implicit tracking.

A.3.2 Implementations

Java Taint Tracking

Erika Chin and David Wagner present a library level taint tracking system by augmenting Java to store and track taint information at the character–level [3]. They then apply it to the Java–based web forum JForum.

The authors addressed several complexities specific to Java. First, Java works directly with character primitives in its implementation. The author's acknowledge

that some taint information is lost in these situations. Second, Java supports string interning which may or may not be used by the application. If it is used, then two separate string objects may refer to the same character array in memory, but one string may be tainted while the other is not. The authors acknowledge that taint information is lost here as well. Finally, Java's strings are encoded in UTF–16, so a single character in a string may actually be stored as two character primitives in the underlying character array of the string object. If this is the case, both character primitives have a separate taint status, but if one primitive of a two–primitive character is tainted, then the other primitive is assumed to be tainted as well.

The authors implementation only tracks taintedness within the string class. They represent taint information for a string object by adding a boolean array called *taintarr*, where *taintarr[i]* corresponds to the taint status of the i^{th} character of the string. Note that *taintarr[i]* may refer to two primitives because of Java's variable length encoding. As an optimization, a null *taintarr* means the entire string is untainted. They did not modify the equals method, so taint status does not affect string equality. They also did not modify the *hashcode()* method nor the string interning methods to reflect taint status. In addition, the methods *replace()* and *setCharAt()* do not propagate taint information because of the difficulties with the character primitive. Instead, these methods preserve the taint status of the original character that is overwritten.

The authors measured the overhead of marking and propagating taint status, but they did not use the taint information to enforce any type of policy. For their tests, they considered all of the functions which access HTTP requests as sources, and all of the string functions which return new strings as propagators. Thus all data obtained from an HTTP request is marked tainted. They note that it is possible for the Servlet to lose some taint information if the developer accesses the HTTP requests as a raw character array (using *getBytes()*, *getChars()*, etc...), however they hypothesize that

this would rarely happen in practice. They tested their scheme with a web server running the JForum application on a machine with only 1GB of RAM, and they made GET and POST requests with 4 client machines. They compared the throughput of the web server (defined as the number of requests processed per second) with taint tracking both enabled and disabled, and they found an overhead of between 0–15% with an average of 5%.

PHP Taint Tracking

Anh Nguyen-Tuong, et al. implemented taint tracking in PHP by augmenting the PHP interpreter, and they used it to enforce a security policy that prevents command–injection and cross–site scripting attacks [16]. Their implementation is similar to the Java implementation. Tracking is done at the character–level, it is deployed in a web application setting, and all data associated with an HTTP request is marked as tainted. Unlike the Java implementation, they also track the source of the taint. Sources include a GET method, POST method, a cookie, a database response, or a session variable. In addition, they take a more conservative approach to trusted sources; they consider data received from a database as untrusted in addition to HTTP requests.

The authors had to address two complexities specific to PHP both of which involved propagating taint information. First, PHP allows string variables to be typecast to integers, and the author's implementation only stores taint information for strings. They acknowledge that this causes taint information to be lost. For example, in the code:

```
$s1 = "143"; // s1 is tainted
$n = $s + 5; // $s is typecast to an integer
```

```
$s2 = "The number is $n"; // s2 is untainted
```

we lose taint information when *$s1* is typecast to an integer. Second, PHP stores session variables in files, so the authors had to modify PHP to store taint information about session variables. They also note that PHP allows developers to replace session storage with their own handlers which would circumvent their handling and thus losing taint information about session variables.

Propagating taint information at the byte granularity requires special handling when only part of a new string is derived from a tainted string. To address this, the authors modify the built–in string functions of PHP to propagate taint. However, developers may use their own functions to build strings in ways that the authors cannot anticipate. To ensure that taint information is not lost, a string returned from a non–built–in function is entirely tainted if the function had a tainted argument.

The authors used their taint tracking to implement a security policy that prevents command–injection and cross–site scripting attacks. They designed the policy to work out–of–the–box by replacing the user's PHP interpreter with their augmented interpreter. To prevent command–injection attacks, they disallowed functions such as exec, *fopen()*, *eval()*, and include from executing if one of their arguments is tainted. In addition, they augmented the functions that can send SQL queries such as *mysql_query* to check the query string for SQL tokens (logical operators, SELECT, OR, etc...) that are tainted. If such a token is found, the query fails. To prevent cross–site scripting attacks, they modify the output functions such as *echo* and *print* to scan output generated using tainted data for potentially dangerous characters. If such data is found, it is either removed or replaced with a safe representative, for example, replacing "<" with "<".

Propagating taint information fared well in the benchmark assessment, achieving a runtime overhead of 3.2%. Printing data and sending SQL queries did not

fair well however, achieving runtime overheads of 45.7% and 76.8% respectively. On the other hand, the augmented interpreter fared well with typical web application requests such as logging in, storing a message in a database, and generating a page from entries in a database, achieving runtime overheads of less than 5% for all benchmarks.

C Taint Tracking

The Java implementation of taint tracking has a number of shortcomings. First, it does not support arbitrary data types. The implementation only supports taint tracking in string objects, and extending this implementation to another data type would require additional modification of the classes and methods that handle the data type. In addition, the authors did not present a method to propagate taint information across primitives that does not require modifying method signatures. Second, it only defends against command injection vulnerabilities. Third, taint information handling must be done in the implementation of the application. For example, the user must write error handling code in the application itself that checks if tainted data will be used in an SQL query and then handles cases where this is true. If an application has many calls to a function that executes an SQL query, each of these calls must be augmented manually by the user.

Chang, et al. proposed an augmented C compiler that implements a security policy defined by the user that is expressed in the Broadway declarative annotation language [2]. Their compiler is a data flow tacking system rather than a taint tracking system because it can track arbitrary properties, called *typestates*, of data. Data is tracked at the byte level by keeping *typestate* information for individual bytes in a tree separate from the bytes. Code is inserted by the compiler that creates and maintains the tree in addition to code that performs the policy checks described by

the security policy. For example, the authors provide a sample security policy in the policy language. It first defines the property taint. It then defines the rule that marks anything returned by the *recv()* function as tainted. Next it defines a rule that says the taint values of the string passed into the function *strdup()* will be propagated to the return value of the function. Finally it defines a rule that prevents the function *printf()* from using a tainted format string.

The authors improve the overall efficiency of their taint tracking implementation by performing an initial static analysis on the input program. The analysis first looks for areas of the program where the policy could potentially be violated. If there are none, then the program is compiled as normal, and a true 0% overhead is achieved. If there are potential violations, then a more complicated analysis is done to determine which areas cannot be proven safe, and these areas are augmented with taint tracking and the policy checks. The additional analysis is comprised of pointer analysis and data flow slicing.

The authors tested their system on server programs and compute–bound programs. For server programs, they compiled pfingerd, Apache, muh, wu-ftpd, and bind with a security policy that prevented a CVE vulnerability specific to the program except for Apache which used a policy that prevented general format string vulnerabilities. The system performed well and was able to detect every vulnerability, it expanded the program binaries by only 0.91% on average, and the response time added was negligible. Similar results were obtained when applying a policy that prevented file disclosure vulnerabilities. For compute–bound programs, they compiled gzip, vpr, mcf, and crafty using the format string vulnerability policy used for Apache. In addition, the authors inserted a format string vulnerability into each of these programs in a place where they argue would represent realistic vulnerabilities. The system performed well on all but gzip which exhibited an overhead of 51.35%.

The authors say that this is due to gzip's complex operations that prevent the static analysis from ruling out potential vulnerabilities.