Debreach: Mitigating Compression Side Channels via Static Analysis and Transformation

Brandon Paulsen[1], Chungha Sung[1], Peter A.H. Peterson[2], and Chao Wang[1]

[1]University of Southern California, Los Angeles, CA, USA
[2]University of Minnesota Duluth, Duluth, MN, USA

Abstract*—Compression is an emerging source of exploitable side-channel leakage that threatens data security, particularly in web applications where compression is indispensable for performance reasons. Current approaches to mitigating compression side channels have drawbacks in that they either degrade compression ratio drastically or require too much effort from developers to be widely adopted. To bridge the gap, we develop Debreach, a static analysis and program transformation based approach to mitigating compression side channels. Debreach consists of two steps. First, it uses taint analysis to soundly identify flows of sensitive data in the program and uses code instrumentation to annotate data before feeding them to the compressor. Second, it enhances the compressor to exploit the *freedom to not compress* of standard compression protocols, thus removing the dependency between sensitive data and the size of the compressor's output. Since Debreach automatically instruments applications and does not change the compression protocols, it has the advantage of being non-disruptive and compatible with existing systems. We have evaluated Debreach on a set of web server applications consisting of 145K lines of PHP code. Our experiments show that, while ensuring leakage-freedom, Debreach can achieve significantly higher compression performance than state-of-the-art approaches.

I. Introduction

Compression is a technique for improving performance, especially in web applications. For example, the DEFLATE [1] compression format in HTTP [2] is used by 70% of the top one million websites [3] because it reduces the size of web content such as HTML, CSS and JavaScript by up to 70-80%. This not only decreases latency and increases throughput, but also reduces energy consumption [4] for battery powered devices. DEFLATE uses two techniques: Huffman coding [5] and LZ77 matching [6], the latter of which is particularly effective for web content. It replaces repeated strings with a *reference* to an earlier copy in the input. For example, if the input is *"Bob is great, Bob is cool"*, the output would be *"Bob is great,* $\langle 14, 7\rangle$*cool"* where the reference $\langle 14, 7\rangle$ is interpreted as *go back 14 bytes and then copy 7 bytes*. Web content has many repeated strings, such as URLs and HTML tags; for example, the string *"wikipedia.org/wiki/"* appears 96 times on Wikipedia's home page, which will be reduced in size by 85%.

Unfortunately, dictionary compression in general, and LZ77 in particular, introduces side channels that can be exploited in *partially chosen-plaintext attacks* [7]–[10]. In such a case, the attacker feeds a *guessed* text to the victim's application, which combines the text with its own sensitive text before giving them to the compressor. When the guessed text matches the sensitive text, the compressed file will be smaller due to LZ77. Since encryption does not hide size, this information can be leaked to an attacker even if the compressed file is encrypted before transmission. Consider the sensitive text *SSN: 123456789* and the guessed texts *SSN: 1234* and *SSN: 1235*. The former will be a complete match, but the latter will not include the last character (5). Since LZ77 references typically have the same size, the former will be one byte smaller.

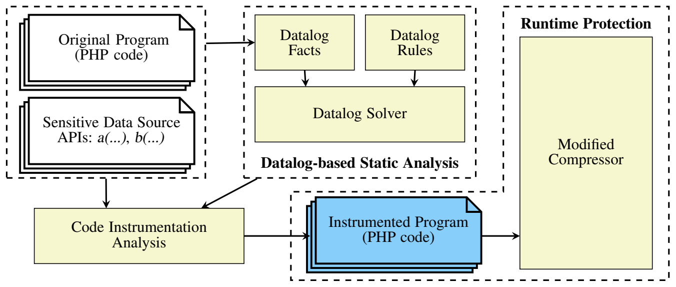

Original Program (PHP code)
Sensitive Data Source APIs: a(...), b(...)
Datalog Facts
Datalog Rules
Datalog Solver
Datalog-based Static Analysis
Code Instrumentation Analysis
Instrumented Program (PHP code)
Runtime Protection
Modified Compressor

Fig. 1: The overall flow of our Debreach method.

Compression side channels were first investigated by Kelsey [7], who found a potential attack against encryption that later was adapted to the real world. For example, Rizzo and Duong [9] proposed an attack named CRIME for the widely-used TLS (successor of the now-deprecated SSL); in response to the risk and due to lack of better solution, the community accepted the solution of disabling TLS compression entirely [11], which is unfortunate. Gluck et al. [8] proposed BREACH, another refinement of Kelsey's technique, which exploits HTTP compression; in response, HTTP/2 altered the compression algorithm to prevent side-channel attacks on header data at the cost of compressibility [12]. Both CRIME and BREACH require a passive eavesdropping point on the encrypted connection to measure the encrypted data size; Vanhoef and Goethem [10], [13], on the other hand, lifted such a requirement by estimating the size through the time taken for a request to complete.

Despite the severity of such security threats, there does not yet exist a general approach that provides sound guarantees about leakage-freedom while maintaining acceptable levels of compression and run-time performance. To fill the gap, we propose Debreach, an approach to mitigating compression side channels in web server applications. Debreach can provide security guarantees about protecting arbitrary web content as opposed to protecting, e.g., only security tokens. Furthermore, it does not require developers to manually identify flows of sensitive data in the program; instead, it uses a static analysis to track the sensitive data flow, and based on the analysis, transforms the server program to allow automated annotation of sensitive data at run time. In addition to automation, the Debreach method provides leakage-free guarantees and high compression performance at the same time.

The overall flow is shown in Fig. 1, where the input consists of the program and a set of sensitive APIs. For example, if an API function retrieves sensitive entries from a database, it will be provided as a sensitive data source. Inside Debreach, a static taint analysis is used to identify flows of sensitive data from sources to sinks, i.e., *echo* statements that construct the HTML file to be sent to the client. The analysis is designed to be sound in that it guarantees to include all sensitive data flows. The result is used to rewrite the program code such that it can, at run time, insert annotations for sensitive data before feeding them to the enhanced compressor.

Our taint analysis uses the general framework of declarative program analysis [14]–[17]. In this framework, the PHP program is first traversed to produce a set of *Datalog facts*, which encode the control and data flow structures of the program as well as the sensitive data sources. The fixed-point computation required by taint analysis is codified in a set of *inference rules*, which are combined with the facts to form the entire Datalog program. We then solve the Datalog program using our Python-based solver which is optimized for solving these rules, the output of which is a sound overapproximation of all sensitive flows from sources to sinks. Finally, we perform a code instrumentation analysis to determine the optimal points to annotate the sensitive data, to ensure that data are properly marked when they reach the compressor. The instrumented program (PHP code), combined with our enhanced DEFLATE compressor, can skip LZ77 and instead use Huffman coding only for the marked data.

In contrast, none of the previous works on mitigating compression side channels can provide soundness guarantees about protecting arbitrary data. The mitigation adopted in HTTP/2 [12], for example, protects only header data but not the payload, while approaches based on randomly masking sensitive data protect only security tokens [18], but not data embedded in JavaScript and HTML [19]. Size-randomizing techniques, while easily applicable, have also been shown to be ineffective [20]. Finally, other approaches that exploit the freedom to not compress [21], [22] either cannot protect arbitrary data [21] or do not provide soundness guarantees [22]. In addition, none of the existing techniques can perform safe compression and at the same time avoid degrading the performance to unacceptable levels. Debreach, in contrast, is fully automated in generating sensitive data annotations and the accuracy is almost as high as annotations created by experts manually.

We have implemented Debreach and evaluated it on a set of server applications consisting of 145K lines of PHP code. The goal is to eliminate the dependence between compression performance and sensitive data since it may be inferred or observed by attackers. Our experiments show that Debreach outperforms both *SafeDeflate*, a state-of-the-art technique that does not even provide leakage-free guarantee, and Huffman coding, another technique that disables LZ77 entirely, in terms of compression performance. Our experiments also show that Debreach prevents side-channel leaks in all applications, including several that *SafeDeflate* cannot prevent.

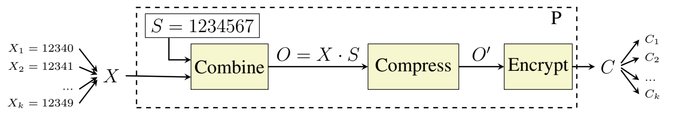


Fig. 2: The adversary model.

To sum up, this paper makes the following contributions:

- We propose Debreach, the first automated approach to mitigating compression side channels for arbitrary web content with security guarantees.
- We implement Debreach in a tool for PHP-based web server applications.
- We demonstrate, through experiments, that Debreach outperforms state-of-the-art techniques in terms of compression performance and security guarantees.

The rest of the paper is organized as follows. First, we motivate our work in Section II by explaining how compression side channels are exploited. Then, we review the DEFLATE compressor in Section III before presenting our enhancement. Next, we present our static analysis in Section IV and code instrumentation in Section V. We present our experimental results in Section VI, review the related work in Section VII, and then give our conclusions in Section VIII.

II. Motivation

In this section, we explain what compression side-channels are and how they are exploited.

A. The Adversary Model

First, we present an adversary model to illustrate the security risks. Suppose an attacker wishes to decrypt part of an encrypted message C with some compressed plaintext O produced by a procedure P, which takes an input string X. Fig. 2 shows the scenario. The attack can be achieved if the following conditions are met. (1) O contains some sensitive string S of interest. (2) O contains some other input string X. (3) P can be executed repeatedly with X chosen by the attacker. (4) P uses some form of dictionary compression on O. (5) The size of the C is visible to the attacker.

Conditions 1, 2 and 4 mean that X and S are always compressed together. Conditions 3 and 5 mean that the attacker can observe how different the compressed sizes are when different choices of X are combined with S to form O.

As an example, consider the sensitive string S=`1234567`. When X is `12344`, LZ77 will encode the first four characters of X with a reference; let this output be C_1. When X is `12345`, the first five characters will be encoded; let this output be C_2. All other content being equal, C_1 will be one byte larger than C_2 because of how X is encoded. In general, the size of C becomes smaller as X becomes more *similar* to S. This information can be exploited to infer the content of S.

```
1  <?php
2  function htmlTag($tg, $data, $attrs)
3  {
4    $ret = "<".$tg." ";
5    $ret .= $attrs;
6    $ret .= ">".$data."</";
7    return $ret.$tg.">";
8  }
9  echo "<html>";
10 $entries = get_addressbook_entries(); //taint source
11 foreach ($entries as $entry) {
12   ...
13   if (strlen($entry->name) > 20)
14     $name = substr($entry->name, 0, 20);
15   else
16     $name = $entry->name;
17   echo htmlTag("a",   //a sink of the tainted data
18        $name,
19        "href='/compose.php?sendto=" .
20        $entry->email . "'");
21 }
22 ...
23 echo "<input type='hidden' value='";
24 echo $_SERVER["QUERY_STRING"];
25 echo "' name='query_string' />";
26 ...
27 echo "</html>";
```

Fig. 3: Compression side channel in a PHP program that creates an HTML page of a user's email addressbook.

B. A Realistic Attack Example

We now show a scenario where the victim uses a webmail, and the attacker wants to know who are in the victim's addressbook. Since the webmail uses the DEFLATE compression followed by encryption to process data passing between the victim's browser and the webmail server, and the attacker may observe the size of the data in transition [7]–[10], there is a side-channel leak. For example, if the attacker is on the same network as the victim and tricks the victim into visiting a malicious web page, he may leverage cookies in the browser to send requests to the server on behalf of the victim.

Fig. 3 shows the server-side PHP code responding to such a request, which generates the user's email addressbook on the returned HTML page. In PHP, the `echo` statements (e.g., at line 17) cause data to be compressed and then encrypted before they are sent to the client.

The sensitive data are retrieved at line 10 using the API function `get_addressbook_entries()`. Each `$entry` is also considered sensitive. Following the data flow, we identify a sink of the tainted data at line 17, which uses the function `htmlTag` (defined at lines 2-8) to construct a string and then feeds it to the `echo`.

Lines 23-25 print a *hidden field* to the HTML, containing the query string of the request URL. Passing back the request URL in a hidden field is common practice in server-generated HTML pages, e.g., to redirect back to that URL after a form's input is handled. In this attack, the hidden field will be exploited by the attacker to infer the sensitive data.

Email addr.	Attacker's guesses		Compressed data
sendto=bob@test.com	Iter. 1	sendto=a	sendto=bob@test.com (23,7)a
sendto=bob@test.com	Iter. 2	sendto=b	sendto=bob@test.com (23,8)
sendto=bob@test.com	Iter. 3	sendto=c	sendto=bob@test.com (23,7)c

Fig. 4: Data containing a sensitive email address and three attacker's guesses lead to different compression sizes.

Now, we relate the example to the adversary model in Section II-A: the PHP program is P, the email address is the sensitive string S, the query string of request URL (hidden field) is the input X, and the HTML is the output O.

The attacker causes the procedure P to execute by making requests to the server with the desired X in the query string. As mentioned earlier, this is possible if the attacker makes requests from the victim's browser, which has the victim's cookies and thus can make the requests appear to be authenticated requests (however, the attacker *will not* see the content of the response HTML because of the *same-origin* policy). Nevertheless, the HTML file size is observable.

To understand how the attack works, consider Fig. 4 as an example. To decrypt an email address, the attacker will attempt to guess it, character by character. Assume that email addresses are composed from the alphabet: $[A-Za-z0-9\_@.]$ and the addressbook contains an entry with the email address `bob@test.com`. For simplicity, also assume it is the only email address to be displayed on the HTML page.

The attacker can *bootstrap* with a known prefix: `sendto=`. To determine the next character, the attacker sends 64 requests to the server, one for each guess character in the alphabet $[A-Za-z0-9\_@.]$ appended to the prefix. The first response has only `sendto=` compressed using the reference `(23,7)`, whereas the second response has the longer string `sendto=b` compressed using the reference `(23,8)`. With the other guess characters, the compressed data size will be the same as that of character `a`. Therefore, `b` is the correct guess.

On each iteration, the attacker determines one character. If the secret has N characters, the attack takes as little as N iterations; this makes it extremely dangerous in practice.

We choose to use the above example because of its simplicity and ease of understanding, but more refined attacks also exist [8], [20]. Nevertheless, the simple example already illustrates the risk of the compression side channel. Given that 70% of the top websites enable DEFLATE compression [3], this is particularly alarming to the security-conscious.

C. Our Mitigation Technique

In Debreach, we take a two-pronged approach. First, we enhance the compressor to exercise the freedom to not compress for any input data surrounded by special markers. Second, we use static analysis and code instrumentation to identify flow of sensitive data through the program and automatically insert special markers. In Fig. 3, for example, it would identify the echo of `htmlTag` at Line 17 as leaky. Then, it would identify instrumentation points in the PHP code (sensitive arguments `$name` and `$entry->email` to `htmlTag`) for generating markers at run time.

TABLE I: Comparing our method to existing approaches.

Method	Arbitrary data?	Leak-free Guarantee?	Fully Automated?	High Compression?
Debreach (our new method)	✓	✓	✓	✓
Keyword-based [22]	✓		✓	
Masking-based [19]		✓		✓
Huffman-only [5]	✓	✓	✓	

Table I compares Debreach with state-of-the-art techniques, which lack in performance, generality, or automation.

1) Keyword-based: Techniques such as *SafeDeflate* [22] utilizes two kinds of keywords: a *sensitive* alphabet (A) and a predefined dictionary (D) of *non-sensitive* strings. Compression of a sequence of characters $L = l_0 l_1 ... l_m$ is allowed only if it does not begin or end with a sensitive character in A, or it matches a string in D. This characterization of sensitive strings is unsound, and provides no security guarantee. For example, a user may configure the alphabet $A = [A-Za-z0-9\_@.]$ to protect emails, but the match *'=bob@test.com'* is still allowed under this configuration. Furthermore, keyword-based techniques degrade the runtime performance and compression ratio substantially. We shall demonstrate both problems experimentally in Section VI.

2) Masking-based: Techniques such as CTX [19] generate a random and reversible *masking* operation and apply it to sensitive data before sending it to the client. On the client side, special JavaScript code must be used as well to undo the masking. The masked data must be enclosed in HTML *<div>* tags with some unique ID attribute. This has two drawbacks. First, it does not work for many applications, including our example in Fig. 3 because inserting *<div>* around *bob@test.com* in the URL would break the link. Fig. 5 shows the example HTML with the email address at Line 3 and hidden field at Line 5. While one could mask the whole HTML tag at Line 3, it would break the initial parsing of HTML. This problem also applies to sensitive data in JavaScript code.

3) Huffman-Only: The naive approach to mitigating compression side channels in DEFLATE is to disable LZ77 and use Huffman coding only. This is guaranteed to be secure because the only information that may be deduced from the compressed file would be the *symbol distribution*, but so far, no attacks have been reported to exploit the symbol distribution. However, this approach has a high performance penalty because, even in the best case, Huffman coding can only reduce a single byte's size by 62.5% (and on average much lower), which is significantly worse than enabling LZ77.

Compared to these existing techniques, Debreach has the advantage of protecting arbitrary web content *soundly* (with a security guarantee) and *more efficiently* (in terms of compression ratio) while requiring no manual effort. In Fig. 3, it would only identify the email addresses as sensitive, and all other web content would be free for compression.

III. Enhancing the Compressor

We first review the state-of-the-art DEFLATE compressor, and then present our enhancement.

```
...
<a href='/compose.php?sendto=bob@test.com'>Bob Anderson</a>
...
<input type='hidden' value='sendto=b' name='query_string'/>
...
```

Fig. 5: HTML response with guess character `b` at line 4.

Algorithm 1: The DEFLATE compression procedure.

```
1: DEFLATECOMPRESS($input$) {
2:     $buf \leftarrow$ LZ77MATCHING($input$);
3:     $output \leftarrow$ HUFFMANCODING($buf$);
4:     return $output$;
5: }
```

A. The Original Compressor

The DEFLATE compressor, shown in Algorithm 1, combines Huffman coding [5] and LZ77 matching [6], and can be found in various open-source libraries including *zlib* [23].

1) Huffman Coding: Given an input composed of symbols from some alphabet, it assigns bit strings to symbols and then encodes the input using these bit strings. It saves space by assigning shorter bit strings to symbols used more frequently. No bit string is a prefix of another bit string, which avoids ambiguity when decoding bit strings back into symbols. Fig. 6 shows an example of applying Huffman coding to part of a popular tongue-twister. There are two columns in the row of Huffman coding: the right shows bit strings assigned to each input symbol, based on the symbol count distribution, and the left shows the encoding result.

2) LZ77 Matching: In dictionary compression, a set of literal strings, called a *dictionary*, is stored in the compressed file, and any input string that matches a dictionary string is encoded as a *reference*. In LZ77 [6], the dictionary is the input itself, and each reference can only point to locations earlier in the input. The dictionary string is called a *match* and the encoded data, or reference, is called the *reproducible extension*. Fig. 6 illustrates LZ77 matching in the third row, where the input has two redundant strings, `she` and `lls`. They are encoded as references $\langle 13, 3\rangle$ and $\langle 10, 3\rangle$, respectively.

Input string to be compressed	she sells seashells	
Output of Huffman-coding only	Encoding	Huffman Codes

	11 100 00 1011 11 00
	01 01 11 1011 11 00
	1010 11 100 00 01 01 11
		e = 00, h = 100
		l = 01, a = 1010
		s = 11, ' '= 1011

Output of LZ77 matching only	she sells sea<13,3><10, 3>	

Fig. 6: Example for Huffman coding and LZ77 matching.

B. Our Modified Compressor

Our modified LZ77-matching procedure includes a preprocessing step, which identifies and removes the annotations of sensitive data (in the form of special tokens). While removing these special tokens, it also computes the necessary metadata that indicate the locations of the sensitive data. Viewing the compressor's input as an array of bytes, the metadata stores, for each index in the input buffer, its forward distance to the next closest region of sensitive data.

The procedure is shown in Algorithm 2, where the metadata `nextTaint[i]` denotes the distance from $input[i]$ to the next region of sensitive data (i.e. $input[i + nextTaint[i]]$ is sensitive). `nextTaint[i]==0` means that $input[i]$ is sensitive. The procedure incrementally and forwardly considers each index i in the input beginning from $i = 0$. When the current index is not sensitive (lines 9-34), we first search the previous input (i.e., $input[0..i - 1]$) for matches (lines 9-22). If a match is found, an LZ77 reference is output, and i is incremented by the length of the match (lines 23-29). Otherwise, i is incremented by 1, and the literal input byte is output (lines 30-33).

To quickly find matches, a dictionary is maintained that records the positions of previously seen strings. The dictionary is updated with every string in the input buffer (lines 25-27 and 31), unless i is in a sensitive region (lines 5-8). The dictionary is then queried for positions of candidates matches for the current index (line 10). For compression ratio efficiency, a minimum match length is defined ($minMatch = 3$ for both Debreach and the original zlib), and for memory efficiency, only strings of length $minMatch$ are stored in the dictionary. However, when searching for a match, we compute the maximum length (lines 12-16), so the best match is *always* found in the previous input buffer.

1) On the correctness: The procedure is correct in that LZ77 matches are not allowed with sensitive data. To prove this, it suffices to show that neither of the two components of an LZ77 match (the match and the reproducible extension) may contain sensitive data. There are four cases to consider where sensitive data may be included in a match, which are illustrated in Fig. 7. Each case represents a state of Algorithm 2, which consists of an input buffer, a region of sensitive data shown in shaded red, a reproducible extension (i and $i+len$), and a match ($matchLoc$ and $matchLoc+len$).

First, observe that we never enter the match-searching block (lines 9-34) when i is sensitive (i.e., $nextTaint[i] == 0$) because of the guard at line 5. This ensures that the reproducible extension cannot start in sensitive data (case 1 in Fig. 7). Next, leveraging the fact that $nextTaint[i]$ is the (forward) distance to the *closest* region of sensitive data (i.e., $input[i]$ to $input[i + nextTaint[i] - 1]$ is *not* sensitive), it also cannot extend into sensitive data (case 2). This is because, when searching for a match (lines 9-22), we set a maximum match length (line 11), which is the minimum of $nextTaint[i]$ and $nextTaint[matchLoc]$. This guarantees the reproducible extension cannot extend into sensitive data.

In addition, a match cannot start in sensitive data (case 3). Notice that the dictionary is only updated at lines 25-27 and 31 with the locations in the reproducible extension or the literal byte, which we just showed cannot not contain sensitive data. Since we only search the dictionary for match locations (line 10), the property then follows. Finally, the match also cannot extend into sensitive data (case 4) since we set a maximum match length (line 11).

The correctness then follows trivially, since the two components of an LZ77 match cannot start in nor extend into sensitive data.

Algorithm 2: Our new LZ77 matching in Debreach.

```
1:  LZ77MATCHING($input$, $nextTaint$) {
2:      initialize $dict$, $output$ to empty
3:      $i = 0$
4:      while ($i < input.length$) {
5:          if ($nextTaint[i] \leq minMatch$) { // skip sensitive data
6:              $output$ += $input[i]$
7:              $i$ += 1
8:          } else { // search for matches
9:              $bestLen = bestDist = 0$
10:             foreach ($matchLoc$ in $dict[input[i..i + minMatch]]$) {
11:                 $maxLen = min\ (nextTaint[i],\ nextTaint[matchLoc])$
12:                 $matchLen = 0$
13:                 while ($input[i + matchLen] ==$
14:                         $input[matchLoc + matchLen]$) {
15:                     $matchLen$ += 1
16:                 }
17:                 $len = min\ (maxLen, matchLen)$
18:                 if ($len > bestLen$) {
19:                     $bestLen = len$
20:                     $bestDist = i - matchLoc$
21:                 }
22:             }
23:             if ($bestLen > minMatch$) {
24:                 $output$ += $\langle bestDist, bestLen\rangle$
25:                 foreach ($j \in i..i + bestLen - minMatch$) {
26:                     $dict[input[j..j + minMatch]] = j$
27:                 }
28:                 $i = i + bestLen$
29:             } else {
30:                 $output$ += $input[i]$
31:                 $dict[input[i..i + minMatch]] = i$
32:                 $i$ += 1
33:             }
34:         }
35:     }
36:     return $output$
37: }
```

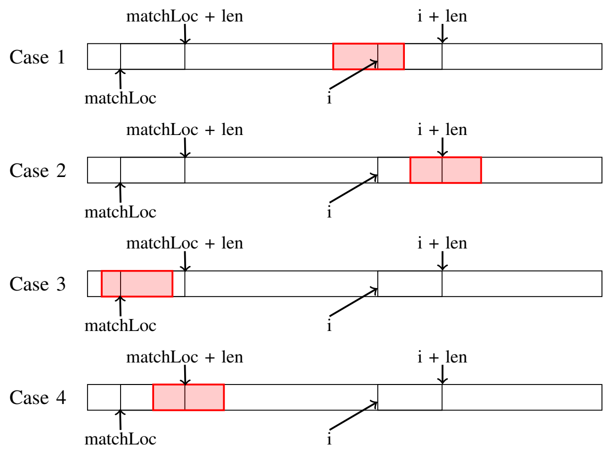


Fig. 7: LZ77: four possible cases for sensitive data.

2) On the security guarantee: Our enhanced compression algorithm guarantees freedom from leaks *due to LZ77* about the *literal content* of the demarcated sensitive data. This follows from the correctness of the algorithm since LZ77 matches cannot contain sensitive data, which implies there can be no dependency (in the behavior of LZ77) between an attacker-controlled plaintext and the literal content of the sensitive data.

However, we do not attempt to eliminate other leaks about other information, such as the length of the sensitive data. In addition, we do not prevent purely size-based side channel leaks, e.g., as in [24]. This is analogous to how a mitigation technique for power-based side channels does not mitigate leaks for timing side channels.

Our algorithm guarantees a form of *non-interference* [25]–[27] about the LZ77 matching procedure. Informally, a prodedure guarantees non-interference if the values of secrets do not influence its behavior in an observable way. Formally stated, let P be our program (Algorithm 2), with public string inputs $X = \{x_1, ..., x_n\}$ and secret string inputs $S = \{s_1, ..., s_m\}$, i.e. $P(X, S)$ is:

$$\texttt{LZ77Matching}(x_1 \cdot s_1 \cdot x_2 \cdot ... \cdot s_m \cdot x_n,\ \mathit{nextTaint})\ ,$$

Then, let $R_P(X, S)$ be the information observable to the attacker, which in our case is the length of the output of P (references have size 1). The non-interference property that Algorithm 2 guarantees is:

$$\forall X, S, S', \text{such that } \forall i\ |s_i| = |s'_i|. \\ R_P(X, S) = R_P(X, S')\ .$$

The above property says that, for any fixed public inputs X, changing the content of the secret variables does not affect the attacker's observations, however changing the length of the secrets may.

IV. Identifying the Sensitive Data

We develop a conservative static analysis to identify the sensitive data. Results of the analysis are then used to perform code instrumentation, to be presented in Section V.

A. Sources of Sensitive Data

Our analysis relies on a set of sensitive APIs that act as tainted sources. In Fig. 3, for instance, the sensitive function is `get_addressbook_entries()`. For server applications targeted by Debreach, the choice of sensitive APIs is often obvious. For example, when the PHP code implements a webmail application, sensitive data are the user's email data and the PHP program must connect to an email provider's IMAP server to retrieve email data. Therefore, the functions that communicate with the IMAP server are sensitive.

The webmail application *NOCC* [28], in particular, uses PHP's IMAP module, and thus built-in functions such as *imap_fetchbody* and *imap_fetchheader* are treated as sensitive API functions. The application named *Squirrelmail* [29], on the other hand, implements its own IMAP communication API; in this case, it uses two functions named *sqimap_run_command* and *sqimap_run_command_list*, which are treated as sensitive API functions.

Other server applications may store sensitive data in a database, for which PHP provides API functions. Therefore, these APIs should be treated as sensitive. For example, the addressbook applications *iAddressbook* [30] and *Addressbook* [31] store sensitive contact information in a database.

Database administration tools such as *Adminer* [32] also have sensitive APIs. For example, *Adminer* [32] is designed with the *select* function used as an API for accessing row data. Thus, for Debreach, considering this as sensitive will achieve the desired security and compression performance.

B. Datalog-based Taint Analysis

We follow a declarative program analysis framework [14]–[17], which first traverses the PHP code to construct a Datalog program, consisting of a set of *facts* and a set of *inference rules*. The Datalog facts are relations that are known to hold in the PHP code. The inference rules define how to deduce new facts from existing facts. Since the analysis can be formulated as a fixed-point computation, in Datalog, the inference rules will be applied repeatedly until all facts are deduced. Consider the example Datalog program below:

$$\begin{array}{ll} \textsc{Edge}(n_1, n_2) & \\ \textsc{Edge}(n_2, n_3) & \\ \textsc{Path}(x, y) & \leftarrow \textsc{Edge}(x, y) \\ \textsc{Path}(x, y) & \leftarrow \textsc{Edge}(x, z),\ \textsc{Path}(z, y) \end{array}$$

The first two lines define facts regarding nodes in a graph: there is an edge from n_1 to n_2 and another edge from n_2 to n_3. The next two lines define the inference rules, saying that (1) if there is an edge from x to y, there is a path from x to y; and (2) Path is transitively closed. Any Datalog engine may be used then to solve the program, the result of which is the set of all pairs that satisfy the Path relation. By querying the result, we know if $\textsc{Path}(n_1, n_3)$ holds.

1) Generating Datalog Facts: First, we construct an inter-procedural control flow graph (ICFG) from the input program where each node in the ICFG corresponds to a program statement. We perform flow- and context-insensitive analyses to determine targets of method calls. We also hard-code data flow information for PHP built-in string functions, such as `substr`, to make our analysis more accurate.

Our analysis handles arrays and fields. For arrays, we create additional program variables for statically known indexes up to one dimension, and we assume unknown indexes could refer to any of these variables. For fields, we take an inexpensive *object-insensitive* and *field-based* approach as proposed by Anderson [33]; it means we do not distinguish the same field names from different objects. The field-based approach is particularly appropriate because, in PHP applications, many classes are only instantiated once in any execution (e.g., a database handle class), or the same field of different heap objects holds similar data (e.g., an email class).

Next, we traverse the ICFG to generate Datalog facts shown as the relations in Fig. 8. The domains are V, the set of variables or objects, S, the set of statements, and F, the set of fields used in a program. We encode control flow as Edge between statements, control dependence as CtrlDep between statements, and leakage-prone branches as UnsafeBranch. The store of a variable or field of an object is encoded as StoreVar or StoreField, and similarly for LoadVar or LoadField. Finally, sensitive API calls and `echo` statements are encoded as Source and Sink, respectively.

EDGE($s_1 : S, s_2 : S$)	Control flow edge from statement s_2 to statement s_1
UNSAFEBRANCH($s_1 : S$)	Branch statement s_1 may cause implicit data flows
CTRLDEP($s_1 : S$, $s_2 : S$)	Statement s_1 is control dependent on s_2
STOREVAR($v_1 : V, s_1 : S$)	Variable or Object v_1 is stored at statement s_1
STOREFIELD($f_1 : F, s_1 : S$)	Field f_1 of an object is stored at statement s_1
LOADVAR($v_1 : V, s_1 : S$)	Variable or Object v_1 is loaded at statement s_1
LOADFIELD($f_1 : F, s_1 : S$)	Field f_1 of an object is loaded at statement s_1
SOURCE($s_1 : S$)	Source of sensitive functions at statement s_1
SINK($s_1 : S$)	Sink (i.e., `echo`) is at statement s_1

Fig. 8: Input relations for our analysis.

2) Generating Inference Rules: Our Datalog rules, shown in Fig. 9, compute (1) the load of tainted data at an `echo` and (2) dependencies for data originating from sensitive APIs.

TAINTEDVARFROM(v_1, s_1, s_1)	$\leftarrow$	STOREVAR(v_1, s_1) $\wedge$ SOURCE(s_1)
TAINTEDVARFROM(v_1, s_1, s_3)	$\leftarrow$	TAINTEDVARFROM(v_1, s_1, s_2) $\wedge$ EDGE(s_2, s_3) $\wedge$ $\neg$ STOREVAR(v_1, s_2)
TAINTEDFIELDFROM(f_1, s_1, s_1)	$\leftarrow$	STOREFIELD(f_1, s_1) $\wedge$ SOURCE(s_1)
TAINTEDFIELDFROM(f_1, s_1, s_3)	$\leftarrow$	TAINTEDFIELDFROM(f_1, s_1, s_2) $\wedge$ EDGE(s_2, s_3)
TAINTED(s_2)	$\leftarrow$	TAINTEDVARFROM(v_1, s_1, s_2) $\wedge$ LOADVAR(v_1, s_2)
TAINTED(s_2)	$\leftarrow$	TAINTEDFIELDFROM(f_1, s_1, s_2) $\wedge$ LOADFIELD(f_1, s_2)
TAINTED(s_3)	$\leftarrow$	TAINTEDVARFROM(v_1, s_1, s_2) $\wedge$ LOADVAR(v_1, s_2) UNSAFEBRANCH(s_2) $\wedge$ CTRLDEP(s_3, s_2)
TAINTED(s_3)	$\leftarrow$	TAINTEDFIELDFROM(f_1, s_1, s_2) $\wedge$ LOADFIELD(f_1, s_2) UNSAFEBRANCH(s_2) $\wedge$ CTRLDEP(s_3, s_2)
TAINTEDVARFROM(v_1, s_1, s_1)	$\leftarrow$	TAINTED(s_1) $\wedge$ STOREVAR(v_1, s_1)
TAINTEDFIELDFROM(f_1, s_1, s_1)	$\leftarrow$	TAINTED(s_1) $\wedge$ STOREFIELD(f_1, s_1)
DATADEP(s_1, s_2)	$\leftarrow$	TAINTEDVARFROM(v_1, s_1, s_2) $\wedge$ LOADVAR(v_1, s_1)
DATADEP(s_1, s_2)	$\leftarrow$	TAINTEDFIELDFROM(f_1, s_1, s_2) $\wedge$ LOADVAR(f_1, s_1)
TAINTEDSINK(s_1)	$\leftarrow$	SINK(s_1) $\wedge$ TAINTED(s_1)

Fig. 9: Datalog rules for data dependency analysis.

Let us walk through the rules in Fig. 9 to better understand our approach. The first and third rules capture when a variable or field is assigned at a sensitive source statement. The relation TAINTEDVARFROM(v_1, s_1, s_2) means v_1 defined in statement s_1 still holds sensitive data at statement s_2, and similarly for TAINTEDFIELDFROM. These two rules initialize the relation for all stored variables/fields at source statements.

The second and fourth rules propagate these relations through control flow graph edges. The difference is that TAINTEDVARFROM can be blocked by a new assignment (STOREVAR), but TAINTEDFIELDFROM cannot be blocked by a new assignment of the field because we do not discriminate different objects for the same field name.

The fifth through eighth rules initialize TAINTED(s_1) when tainted data is used either explicitly or implicitly, meaning s_1 loads tainted variables. Then, the ninth and tenth rules create new TAINTEDVARFROM and TAINTEDFIELDFROM relations for the stored variables at tainted statements. The next two rules infer DATADEP, where DATADEP(s_1, s_2) means s_1 data-depends on s_2, which occurs when tainted data is propagated from s_1 to s_2 and loaded at s_2. Finally, TAINTEDSINK represents when tainted data is used at a SINK statement, i.e., it may be sent through an `echo` statement.

Our approach shares the limitations of other static analysis tools for dynamically typed languages such as PHP, e.g., unsoundness in the presence reflection [34]. However, for the benchmarks used in our experiments, we have confirmed such language features did not affect soundness of our analysis.

C. Implicit Flows

The key to high compression performance is proper tracking of implicit data flows. Since web applications are string-building programs at their core, tainted variables are frequently used in branch conditions. However, naively tracking all these implicit flows would result in over-tainting and low compression performance. Instead, we develop sufficient conditions under which implicit flows can be ignored. Our conditions label a branch as safe if all of the atoms of its predicate are:

- a variable (e.g., `if ($var)`);
- comparing a variable to a hard-coded value;
- comparing to the length of variable; or
- checking the type of a variable.

We do not attempt to protect Boolean variables from implicit data flows, for two reasons. First, even if we protect them during compression, they may still be revealed through purely size-based side channels (i.e., not due to compression). Second, protecting them would prevent us from ignoring frequently-occurring, performance-critical branches that do not affect our security guarantee (specifically the first criterion above).

D. Security Guarantee

The guarantee we aim to provide is that an attacker cannot discover *literal* content of secret *string type* data. For example, the branch at line 13 in Fig. 3 can be ignored, since it may only reveal that the length of the secret was greater than 20 characters. Conversely, the case we *do* care about is when a dynamically determined non-sensitive string (not originating from a source) is compared with a sensitive string and then eventually compressed, for example:

```
$tainted = sensitive_source();
$untainted = $_GET["cgi_param"];
if ($tainted == $untainted)
        echo $untainted
```

Our analysis provides the security guarantee described above given two assumptions. First, we assume that the truth value of dangerous predicates such as the one above do not become associated with the branches we ignore. For example, we assume a dangerous predicate is not first assigned to a variable, and subsequently used in a branch. Second, we assume tainted data flows do not depend on dynamic features such as reflection.

V. INSTRUMENTING THE SERVER PROGRAM

We instrument the program to allow it to generate annotations of sensitive data at run time, prior to the compression.

A. Annotations

Annotations of sensitive data are special markers inserted into the tainted string value. During the execution, the program randomly generates a nonce of arbitrary length, and uses it to enclose the sensitive string. For instance, in Fig. 3, it would wrap the (sensitive) argument at line 18 with markers, e.g., `"DBR{" . $name . "}DBR"`, where `DBR` is the nonce.

Algorithm 3: Computing the instrumentation points

```
1:  INSTRANALYSIS($tsinks$, $DDG$) {
2:    $instr\_pts = [\,]$
3:    for each ($sink \in tsinks$) {
4:      $ctx$ = function of $sink$
5:      $instr\_pts$ += FINDINSTR($sink$, [ ], $ctx$, $DDG$)
6:    }
7:    return $instr\_pts$
8:  }
1:  FINDINSTR($cur$, $visited$, $ctx$, $DDG$) {
2:    $preds$ = immediate predecessors of $cur$ in $DDG$
3:    if ($preds$ == [ ])
4:      return [$cur$]
5:    $preds = \{p \in preds | p \notin visited\}$
6:    $visited$ += $preds$
7:    if ($preds$ == [ ])
8:      return [ ]
9:    else if (for all $p \in preds$, ISSAFE($p$, $DDG$) is $true$) {
10:     $instr\_pts = [\,]$
11:     for each ($p \in preds$)
12:       $instr\_pts$ += FINDINSTR($p$, $visited$, $ctx$)
13:     if (any $pt \in instr\_pts$ is not in $ctx$)
14:       return [$cur$]
15:     else
16:       return $instr\_pts$
17:   }
18:   else
19:     return [$cur$]
20: }
```

B. *Efficient Code Instrumentation*

Naively, we could annotate all tainted variables used at an `echo`, however this may result in the amount of annotated data to become unacceptably large and degrade the compression ratio, e.g., in Fig. 3, where sensitive data in `$entry` are combined with non-sensitive HTML tags.

Naively annotating the tainted source is also problematic because markers inserted into the string may affect subsequent manipulations, e.g., in Fig. 3, where an entry's name is truncated if it is longer than 20 characters (lines 13-16) using the `strlen` and `substr` operations. Clearly, inserting markers before these function calls changes their semantics.

To avoid both problems, we perform an analysis to determine the best instrumentation point. To this end, we consider both *performance*, i.e., how much compression to maintain, and *safety*, i.e., not breaking the program semantics.

Our analysis in Algorithm 3 takes as input the tainted echos and the data dependence graph (DDG) computed earlier. On each tainted echo, FINDINSTR is called to find a set of statements (and variables) to instrument. FINDINSTR performs a backward search along data dependence edges until it encounters a stopping condition, which indicates the current statement is an instrumentation point.

There are three stopping conditions, namely: (a) the current statement has no predecessors, indicating we reached the taint source (line 3), (b) we have already visited all of the statement's predecessors, indicating we have *covered* it with another instrumentation point (lines 6-7), or (c) some predecessor P is unsafe (line 9).

A statement is unsafe if one of two conditions hold: (1) it contains an operation that may be broken by inserted annotation, or (2) it may affect another statement that meets condition (1). The check for condition (1) relies on a whitelisting based approach, where all statements on the whitelist are guaranteed to not be affected by inserting an annotation. Generally speaking, condition (1) will *not* be satisfied (i.e., the statement is safe) if a statement consists of only assignment, concatenation, or PHP built-in functions that can never be affected by inserted annotation.

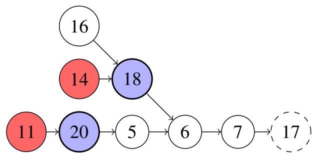


Fig. 10: Data dependence graph of motivating example.

In Fig. 3, statements in `htmlTag` are safe because they only contain string concatenations, but lines 13 and 14 are not safe because annotations may break the called functions. In addition, line 11 is not safe because 13 and 14 depend on it.

Our analysis also limits itself to annotating within the context (i.e., a function or top-level script code) of the tainted sink (lines 13-16). This is to reduce over-annotating. For example, real applications make use of utility functions for generating HTML, similar to the `htmlTag` function in Fig. 3, and they are used frequently with both sensitive and non-sensitive data. Annotating inside these functions would likely degrade compression performance.

We now return to the example in Fig. 3, whose DDG is shown in Fig. 10. Nodes are labeled by line number, the unsafe nodes for instrumentation are filled with red, and the tainted echos are shown with a dashed circle. During code instrumentation, we would step backward from node 17 and branch at node 6. While considering node 18 along the top branch, the safety check at line 9 of FINDINSTR would fail because one of its predecessors (node 14) is unsafe. When we reach node 20 along the bottom branch, the safety check would fail again because node 11 is unsafe. So we would annotate the variables at line 18 and 20 (blue nodes in the graph).

VI. EXPERIMENTS

We have implemented Debreach using a combination of Java, Python, and Datalog. It requires the user to provide the top-level directory of the PHP code and a set of sensitive API functions, and then generates instrumented PHP code. We leverage *joern-php* [35] to extract control-flow and def/use Datalog facts from the PHP code. We extend *joern-php* to handle arrays, fields, objects, globals, and pass-by-reference parameters. Our static analysis is implemented using 2K lines of Java code, 1.6K lines of Python code, which includes our Datalog solver implemented in Python.

We conducted experiments to answer three questions:

- Are the static analysis and instrumentation components in Debreach efficient in handling real web applications?
- Can Debreach achieve high compression performance? In particular, how does it compare to state-of-the-art techniques?
- Can Debreach eliminate the actual side channels?

TABLE II: Characteristics of the benchmark applications.

Application	Lines of Code	Requested Page	Response HTML Size (KB)
Squirrelmail [29]	55,698	compose email	3
		login	2
		preferences	5
		view email	9 - 29
		view inbox	18 - 19
NOCC [28]	17,610	compose email	8
		view email	24 - 102
		view inbox	35 - 37
		login	6
		preferences	16
Adminer [32]	37,330	edit row	2 - 5
		insert item	2 - 4
		login	2
		table data	11 - 66
		table structure	6 - 7
iAddressbook [30]	16,690	addressbook	14
		contact info	23 - 24
		edit contact	48
		login	2
		new contact	33
Addressbook [31]	17,907	contact info	5
		edit contact	11
		login	4
		new contact	3
		addressbook	556

We answer the first two questions by comparing the compression ratios of Debreach to the keyword-based *SafeDeflate* [22], *Huffman-Only* [5], and an *Oracle* version of Debreach, which represents the practical limit of Debreach.

To produce the *Oracle* version of Debreach, we first instrument the application with Debreach and then manually inspect each instrumentation point to decide if it can be optimized. If, for example, the instrumentation was due to a false-positive tainting, we remove it. If it was a true-positive, but a better instrumentation exists (that reduces the amount of tainted data), we change it to the better one.

As for the third research question, we note that Debreach guarantees to eliminate the compression side channel. Nevertheless, it is still informative to demonstrate on real applications. Thus, first, we show that leaks indeed exist and can be exploited for some applications. Then, we show that leaks no longer exist in Debreach-instrumented versions.

A. Experimental Setup

Our benchmark consists of five server applications with 145K lines of PHP code in total. The applications fall into three categories: webmail, database administration, and addressbook. Our main selection criteria is that Debreach has a practical use case for the application. The characteristics are summarized in Table II, where Columns 1 and 2 show the name and number of lines of code. For each application, Column 3 shows the five pages chosen for experiment; they are web pages that users are likely to visit. Column 4 shows the size of the corresponding HTML response. Response size can vary because of the application's dynamic content. For example, email bodies can vary widely in size.

To compare against the keyword-based *SafeDeflate*, we need to configure its required *sensitive alphabet* and the dictionary of allowed strings, as described in the original publication [22]. For *Huffman-Only*, we used the algorithm implemented in *zlib* [23].

Next, we describe the applications in detail, and justify how we configure and compare the different tools.

1) Squirrelmail and NOCC: These are webmail applications, where the user may want to prevent email data from being leaked through the compression side channel. In both applications, email data is retrieved from an IMAP server, so for Debreach, we choose functions that communicate with the IMAP server as sensitive API functions. For *SafeDeflate*'s alphabet, we include alphanumerics, space, comma, and period to protect natural language in email bodies, and we include '/', ':', '%', '&', '=', and '?' to protect links. In addition, NOCC supports HTML in emails, so we add in '<', '>', double quote, and single quote. *Squirrelmail* does not support HTML, so we exclude these characters. For email addresses we include '@' and '_'. While the alphabet provides protection in *most* cases, email bodies allow nearly any printable ASCII character, so *SafeDeflate* may still have leaks.

2) Adminer: This is a database administration (DBA) tool, where the user may want to protect the rows of the database tables from being leaked. Since *Adminer* is designed with a function *select* that is used as an API for accessing row data, for Debreach, we mark this as the sensitive API. We populate the database with a standard SQL dataset that resembles an employee database for a company. The table's data is made of alphanumerics and the hyphen character, so we configure *SafeDeflate* with this alphabet.

3) Addressbook and iAddressbook: These are open-source addressbook applications, where the user may want to prevent the stored contact information from being leaked. Both applications use a database for storing this information, so we taint the query functions for both cases. We populate the addressbook with 500 contacts including names, emails, phone numbers, addresses, jobs, and titles generated using the *faker* Node.js library. Accordingly, we configure *SafeDeflate* with alphanumerics, '-', '@', '.', and '_'.

B. Results of Analysis and Instrumentation

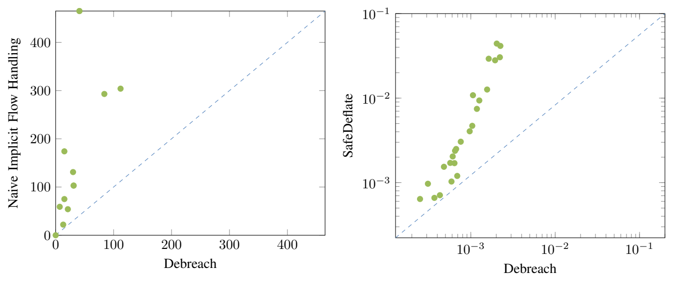
First, we measure the performance of Debreach's static analysis and code instrumentation components. The experiments were conducted on an Ubuntu 16.04 machine with 12GB of RAM and an Intel i7 processor.

Table III shows the results. Columns 1-3 show the application name, page used in experiment, and total number of echos involved. Columns 4-5 show the number of tainted echos and instrumentation points for *oracle*. Columns 6-8 show the results of Debreach, including the number of tainted echos, the number of instrumentation points, and the analysis time.

The results show that Debreach is close to *oracle* in identifying tainted echos and instrumentation points. For most pages there is $<$15% false-positive rate in tainted echos. The majority of false positives are due to implicit data flows. A common case we observe are branches that compare sensitive data to some configuration parameter pulled from a source that we cannot prove to be non-sensitive. In addition, we show in Fig. 11a the benefit of our implicit flow rules (Section IV-C), without which there would be 100's of false positives.

TABLE III: Performance of Debreach's analysis and instrumentation. For NOCC, etc., all five pages are the same (*).

App	Page	Total Echos	Oracle		Debreach (new)		
			Tainted Echos	Instr. Points	Tainted Echos	Instr. Points	Time (s)
Squirrelmail	comp-e	96	10	15	15	17	159
	login	17	0	0	0	0	152
	pref.	58	13	12	13	12	151
	view-e	85	6	12	7	9	156
	view-i	99	12	36	21	37	152
NOCC	*	2423	97	107	112	114	167
Adminer	*	809	26	44	30	48	99
iAddressbook	*	763	83	104	84	106	129
Addressbook	addr-b	220	30	59	31	65	78
	c-info	228	14	19	15	19	72
	edit-c	521	40	43	41	44	69
	login	220	30	59	31	65	78
	new-c	521	40	43	41	44	69



(a) (b)

Fig. 11: (a) Tainted echos with and without Debreach's implicit flow. (b) Execution time (s): Debreach vs. *SafeDeflate*.

As for the time taken to complete the analysis and instrumentation, Column 8 in Table III shows that Debreach takes < 3 minutes in all cases. Furthermore, it is scalable in handling real applications.

C. Performance of Mitigated Application

Next, we evaluate the Debreach-instrumented applications. We focus on the overhead in execution time and the change in compression ratio. We present comparisons to other approaches as scatter plots. Points above the dashed blue-line indicate Debreach outperforms the competing approach.

1) Execution Time: We measure the overhead of our instrumented PHP code and modified compressor separately, by comparing with the original (and leaky) PHP code and original *zlib* compressor [23]. The results are shown in Table IV. Columns 3-5 compare the time in seconds of both the original and instrumented PHP code. Columns 6-8 compare the time in milliseconds of the original and modified compressor code. We average the PHP code time over 100 executions, and the compressor code over 1000 executions because it is a much faster process and therefore affected more by noise.

For 22 of the 25 pages, the runtime overhead in PHP is <3%, which is negligible. This is because the time is generally dominated by network or database accesses, which is also why the overhead is occasionally negative, since the instrumentation overhead is small. In 24 out of 25 pages, the original and modified compressors are within a $1/10^{th}$ of millisecond of each other, and for the remaining page, Debreach is faster. All compressor results are statistically

TABLE IV: Runtime overhead of PHP code with compression.

App	Page	PHP-script Running Time (s)			Compression Time (ms)		
		Orig.	Instr.	Overhead	Orig.	Modif.	Overhead
Squirrelmail	comp-e	0.00082	0.00082	0.00%	0.1115	0.1248	11.88%
	login	0.00036	0.00036	0.00%	0.0428	0.0485	13.41%
	pref.	0.00070	0.00068	-2.86%	0.0830	0.0702	-15.49%
	view-e	0.91540	0.98640	7.76%	0.2090	0.1594	-23.74%
	view-i	1.25098	1.25102	0.00%	0.2556	0.2275	-11.00%
NOCC	comp-e	0.57718	0.55184	-4.39%	0.0234	0.0258	10.62%
	view-e	3.45185	3.55438	2.97%	0.7342	0.6834	-6.91%
	view-i	6.61984	6.68215	0.94%	0.5270	0.5171	-1.88%
	login	0.00067	0.00067	0.00%	0.1072	0.1172	9.34%
	pref.	0.59901	0.64006	6.85%	0.2428	0.2649	9.08%
Adminer	edit-r	0.96962	0.97202	0.25%	0.1095	0.1231	12.41%
	isrt-i	0.00212	0.00214	0.94%	0.0978	0.1143	16.87%
	login	0.00052	0.00053	1.92%	0.0303	0.0331	9.42%
	tbl-da	0.46160	0.46117	-0.09%	0.5788	0.6129	5.90%
	tbl-st	0.00240	0.00247	2.92%	0.0893	0.1002	12.20%
iAddressbook	addr-b	0.08181	0.08182	0.01%	0.2881	0.3128	8.58%
	c-info	0.08213	0.08234	0.26%	0.3768	0.4248	12.72%
	edit-c	0.08256	0.08244	-0.15%	0.6576	0.7498	14.01%
	login	0.00078	0.00076	-2.56%	0.0325	0.0346	6.49%
	new-c	0.08189	0.08195	0.07%	0.6465	0.7249	12.13%
Addressbook	c-info	0.00195	0.00198	1.54%	0.1262	0.1423	12.79%
	edit-c	0.00209	0.00213	1.91%	0.1982	0.2399	21.03%
	login	0.00092	0.00094	2.17%	0.0906	0.1009	11.38%
	new-c	0.00191	0.00195	2.09%	0.1144	0.1345	17.53%
	addr-b	0.16089	0.17013	5.74%	7.8697	4.9878	-36.62%

* NOCC makes many connections to the email server and each connection latency varies every time, which explains the long execution time and big performance difference.

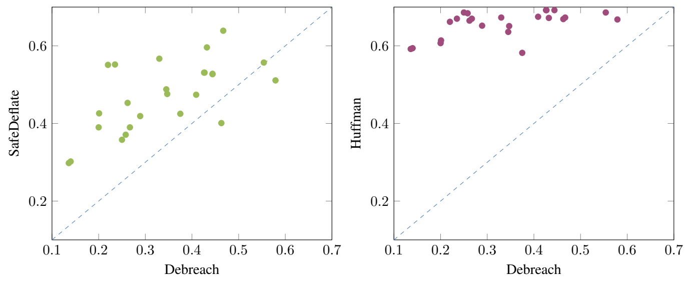


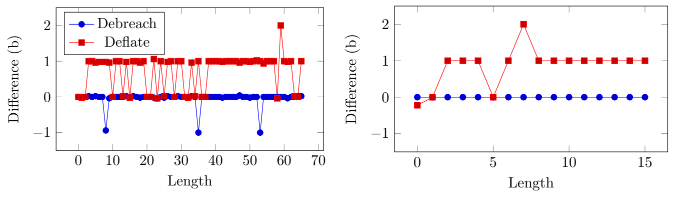
Fig. 12: Comparison of compression ratios: Debreach vs. *SafeDeflate* (left) and Debreach vs. Huffman-Only (right).

significant with $p < 0.00001$. Debreach is faster when the amount of sensitive data is large because Huffman coding is much faster than searching for LZ77 matches. Finally, we compare Debreach with *SafeDeflate* in Fig. 11b and observe that Debreach is 2-5 times faster.

2) Compression Ratio: Fig. 12 compares the compression ratios of Debreach, *SafeDeflate* and *Huffman-Only*. Here, compression ratio is computed as $\frac{compressed\ size}{original\ size}$. The x-axis shows the compression ratio for Debreach, and the y-axis shows the other. We note again here that *SafeDeflate* does not even provide security guarantees.

Debreach is on average 15.6% better than *SafeDeflate*. This is because *SafeDeflate* assumes any region of data composed of sensitive alphabet characters is tainted, but non-sensitive data invariably contain characters from the alphabet, so *SafeDeflate* over-approximates. Debreach, in contrast, is almost always more accurate. For example, login and preference pages either do not call sensitive API functions, or their results do not flow into HTML response, which can be identified by Debreach. Compared to the oracle, Debreach is on average within 2.5%.

The reliance of *SafeDeflate* on a sensitive alphabet is especially problematic in NOCC, where sensitive data may contain HTML. For example, on the inbox view page, Debreach



(a) Squirrelmail (b) Adminer

Fig. 13: Side-channel leak measurements in two case studies.

is able to precisely taint only the email header information such as the sender/receiver email addresses, the email subject lines, the sending date, etc. This accounts for a relatively small portion of the data on the inbox page. Unfortunately, *SafeDeflate* determines full HTML tags and URLs (which make up the majority of any HTML page) are tainted as well. This leads to poor performance (more than 25% worse in four of five pages).

D. Security Evaluation

Now, we report the exploitable leaks in two applications based on what has been described in Section II-A. That is, the adversary is a man-in-the-middle and the victim visits an adversary-controlled web page. For each leak, we identify a target secret to extract (for *Squirrelmail*, it is one arbitrarily chosen subject line from the inbox; for *Adminer*, it is a credit card number stored in the database).

Fig. 13 shows the ability to guess the $(n+1)^{th}$ byte of a target secret given that the attacker knows the first n bytes. The x-axis is the length n of the known prefix, and the y-axis is the difference between the compressed sizes with correct and incorrect guesses of the $(n+1)^{th}$ byte. Here, a difference near one or greater indicates the leak is exploitable.

In both *Squirrelmail* and *Adminer*, there is a strong potential for the application to be exploited once the adversary can determine the first 3 bytes of the secret. After Debreach is applied, however, the difference becomes less than or equal to zero, which means the exploit is no longer possible. In *Squirrelmail*, the correct guess is occasionally larger than incorrect guesses since the encodings of Huffman coding changes, which explains the negative difference. Therefore, this does not leak information.

VII. Related Work

We have already reviewed techniques that are the most closely related to our work, including various attacks that exploit the compression side channel.

Some components of compression side channel attacks have been leveraged in other types of attacks. For example, the timing of cross-site requests have been exploited to reveal private information in web applications [36]–[38]. Chosen plaintext attacks have been previously used to decrypt data encrypted using CBC-mode encryption ciphers [39]–[41]. Recent work has been done on analyzing implementations of compression routines [42]–[44]. Zhou et al. [43], in particular, use *approximate* model counting to estimate leakage, and are able to apply their approach to a real-world compressor. However, none of these works can mitigate the compression side channel in a server application, like we do.

Besides compression, which is a relatively new source of exploitable side channel information, other sources of side channel information have been studied in the past such as timing [45]–[49], cache behavior [50]–[52], and power [53]–[56]. However, attacks and mitigations for these side channels differ fundamentally from compression side channels.

Datalog-based program analysis has been applied in many domains. For example, Whaley and Lam [14] used this framework to perform context-sensitive alias analysis in Java programs. Livshits and Lam [57] and Naik et al. [58] used similar techniques to detect security errors and data-races. Bravenboer and Smaragdakis [59] used it to perform points-to analysis. Sung et al. used it to improve automated testing of JavaScript-based web applications [60] and semantic diffing of concurrent programs [61]. However, none of these prior works used Datalog to mitigate compression side channels.

We also note here that we tried to use off-the-shelf solvers such as *Souffle* [62] to solve our Datalog programs. While *Souffle* finished in all cases, we found that our Python-based solver finished much faster in some cases. This is because our solver is geared toward iterative data-flow analysis (such as our taint analysis) and only supports our rule set, whereas *Souffle* is general-purpose and supports arbitrary rule sets. An interesting research direction would be optimizing general-purpose Datalog solvers such as *Souffle* for iterative data-flow analysis.

Finally, while not the focus of our work, there is a significant body of work focusing on the static analysis of PHP programs. For example, Xie and Aiken [63] are among the first to statically analyze PHP to detect SQL injection vulnerabilities. Since then, many have proposed techniques to model features such as aliasing [64], built-in functions [65], second-order data flows [66], object injection [67], and client-side call graphs [68]. Most recently, Alhuzali et al. [69] combine static and dynamic analyses to synthesize exploits such as SQL injection and cross-site scripting attacks.

VIII. Conclusions

We have presented a *safe* and *efficient* compressor-level approach to mitigating compression side channel attacks. Our approach is based on static taint analysis to safely find tainted sinks and efficient code instrumentation techniques to instrument proper program points. It gives a server application the ability to automatically generate annotations of sensitive data at run time. Moreover, it is fully compatible with existing platforms. We have implemented our approach in the software tool for PHP-based server applications and showed that our approach is both efficient compared to state-of-the-art mitigation techniques, and can prevent leaks on a set of real-world applications, while having minor performance overhead.

Acknowledgments

This work was partially funded by the U.S. National Science Foundation (NSF) under the grant CNS-1617203 and Office of Naval Research (ONR) under the grant N00014-17-1-2896.

References

[1] P. Deutsch, "Deflate compressed data format specification version 1.3," RFC Editor, Tech. Rep., 1996.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," RFC Editor, Tech. Rep., 1999.

[3] W3Techs. (2017) Usage of gzip compression for websites. [Online]. Available: https://w3techs.com/technologies/details/ce-gzipcompression/all/all

[4] K. C. Barr and K. Asanović, "Energy-aware lossless data compression," *ACM Transactions on Computer Systems*, vol. 24, no. 3, pp. 250–291, 2006.

[5] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098 – 1101, 1952.

[6] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, May 1977.

[7] J. Kelsey, "Compression and information leakage of plaintext," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, J. Daemen and V. Rijmen, Eds., 2002, vol. 2365, pp. 263–276.

[8] Y. Gluck, N. Harris, and A. Prado, "BREACH: Reviving the CRIME attack," 2013, black Hat USA.

[9] T. Duong and J. Rizzo, "The CRIME attack," 2012, ekoparty. [Online]. Available: https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit

[10] M. Vanhoef and T. V. Goethem, "HEIST: HTTP encrypted information can be stolen through TCP-windows," in *Black Hat USA*, 2016.

[11] J. Salowey. (2014, mar) Confirmation of consensus on removing compression from TLS 1.3. [Online]. Available: https://www.ietf.org/mail-archive/web/tls/current/msg11619.html

[12] R. Peon and H. Ruellan, "HPACK: Header compression for HTTP/2," RFC Editor, RFC 7541, 2015. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7541.txt

[13] T. Van Goethem, M. Vanhoef, F. Piessens, and W. Joosen, "Request and conquer: Exposing cross-origin resource size." in *USENIX Security Symposium*, 2016, pp. 447–462.

[14] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *ACM SIGPLAN Notices*, vol. 39, no. 6, 2004, pp. 131–144.

[15] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries," in *ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2005, pp. 1–12.

[16] B. Livshits and M. Lam, "Finding security vulnerabilities in java applications with static analysis." in *USENIX Security Symposium*, vol. 14, 2005, pp. 18–18.

[17] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *ACM Symposium on Foundations of Software Engineering*, 2015, pp. 462–473.

[18] L. Pomfrey, "django-debreach," https://github.com/lpomfrey/django-debreach, 2018.

[19] D. Karakostas, A. Kiayias, E. Sarafianou, and D. Zindros, "CTX: Eliminating BREACH with context hiding," in *Black Hat EU*, 2016.

[20] D. Karakostas and D. Zindros, "Practical new developments on BREACH," in *Black Hat Asia*, 2016.

[21] J. Alawatugoda, D. Stebila, and C. Boyd, "Protecting encrypted cookies from compression side-channel attacks," in *International Conference on Financial Cryptography and Data Security*, 2015, pp. 86–106.

[22] M. Zieliński, "SafeDeflate: compression without leaking secrets," Cryptology ePrint Archive, Report 2016/958, 2016.

[23] J.-l. Gailly and M. Adler, "zlib," 2013. [Online]. Available: zlib.net

[24] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, "Sidebuster: automated detection and quantification of side-channel leaks in web application development," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 595–606.

[25] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 53–70.

[26] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[27] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*. IEEE, 1982, pp. 11–11.

[28] T. Gerundt, O. Heil, and S. Mazeland, "Nocc," 2018. [Online]. Available: nocc.sourceforge.net

[29] "Squirrelmail," 2011. [Online]. Available: squirrelmail.org

[30] "iaddressbook," 2017. [Online]. Available: iaddressbook.org

[31] "Addressbook," 2017. [Online]. Available: sourceforge.net/projects/php-addressbook

[32] "Adminer," 2018. [Online]. Available: www.adminer.org

[33] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[34] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[35] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of PHP application vulnerabilities," in *IEEE European Symposium on Security and Privacy*, 2017, pp. 334–349.

[36] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1382–1393.

[37] N. Gelernter and A. Herzberg, "Cross-site search attacks," in *ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1394–1405.

[38] A. Bortz and D. Boneh, "Exposing private information by timing web applications," in *International Conference on World Wide Web*, 2007, pp. 621–628.

[39] G. V. Bard, "A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL," in *SECRYPT*, 2006, pp. 99–109.

[40] A. Joux, G. Martinet, and F. Valette, "Blockwise-adaptive attackers revisiting the (in) security of some provably secure encryption modes: CBC, GEM, IACBC," in *Annual International Cryptology Conference*, 2002, pp. 17–30.

[41] "CVE-2011-3389." Available from MITRE, CVE-ID CVE-2014-0160, 2011. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2011-3389

[42] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan, "String analysis for side channels with segmented oracles," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 193–204.

[43] Z. Zhou, Z. Qian, M. K. Reiter, and Y. Zhang, "Static evaluation of noninterference using approximate model counting," in *IEEE Symposium on Security and Privacy*, 2018.

[44] Q.-S. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, "Synthesis of adaptive side-channel attacks," in *IEEE Computer Security Foundations Symposium*, 2017, pp. 328–342.

[45] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *ACM SIGPLAN Notices*, vol. 52, no. 6, 2017, pp. 362–375.

[46] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 875–890.

[47] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *International Symposium on Software Testing and Analysis*, 2018, pp. 15–26.

[48] T. Brennan, S. Saha, T. Bultan, and C. S. Pasareanu, "Symbolic path cost analysis for side-channel detection," in *International Symposium on Software Testing and Analysis*, 2018, pp. 27–37.

[49] M. Wu and C. Wang, "Abstract interpretation under speculative execution," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 802–815.

[50] G. Barthe, B. Köpf, L. Mauborgne, and M. Ochoa, "Leakage resilience against concurrent cache attacks," in *International Conference on Principles of Security and Trust*, 2014, pp. 140–158.

[51] C. Sung, B. Paulsen, and C. Wang, "CANAL: A cache timing analysis framework via LLVM transformation," in *IEEE/ACM International Conference On Automated Software Engineering*, 2018, pp. 904–907.

[52] S. Guo, M. Wu, and C. Wang, "Adversarial symbolic execution for detecting concurrency-related cache timing leaks," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 377–388.

[53] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated verification of software power analysis countermeasures," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2013, pp. 293–310.

[54] H. Eldib, C. Wang, and P. Schaumont, "Formal verification of software countermeasures against side-channel attacks," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 11:1–24, 2014.

[55] J. Zhang, P. Gao, F. Song, and C. Wang, "SC-Infer: Refinement-based verification of software countermeasures against side-channel attacks," in *International Conference on Computer Aided Verification*, 2018, pp. 157–177.

[56] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 590–601.

[57] B. Livshits and M. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *USENIX Security Symposium*, 2005.

[58] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 308–319.

[59] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2009, pp. 243–262.

[60] C. Sung, M. Kusano, N. Sinha, and C. Wang, "Static DOM event dependency analysis for testing web applications," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 447–459.

[61] C. Sung, S. K. Lahiri, C. Enea, and C. Wang, "Datalog-based scalable semantic diffing of concurrent programs," in *IEEE/ACM International Conference On Automated Software Engineering*, 2018, pp. 656–666.

[62] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in Datalog," in *International Conference on Compiler Construction*, 2016, pp. 196–206.

[63] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages." in *USENIX Security Symposium*, vol. 15, 2006, pp. 179–192.

[64] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Workshop on Programming languages and analysis for security*, 2006, pp. 27–36.

[65] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis." in *Network and Distributed System Security Symposium*, 2014.

[66] J. Dahse and T. Holz., "Static detection of second-order vulnerabilities in web applications." in *USENIX Security Symposium*, 2014, pp. 989–1003.

[67] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in PHP: Automated pop chain generation," in *ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 42–53.

[68] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Building call graphs for embedded client-side code in dynamic web applications," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 518–529.

[69] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and scalable exploit generation for dynamic web applications," in *USENIX Security Symposium*, 2018, pp. 377–392.