G-DBREACH Attacks: Algorithmic Techniques for Faster and Stronger Compression Side Channels

Britney Bourassa[1], Yan Michalevsky[2], and Saba Eskandarian[1]

[1] University of North Carolina at Chapel Hill
[2] Spacecoin.xyz

Abstract. Compression side channel attacks target systems that compress data before encrypting it. By taking advantage of the fact that compression causes message lengths to be connected to message contents, they enable attackers to exfiltrate encrypted data without access to the secret keys.
This paper introduces new techniques for compression side channel attacks on databases. We show how compression side channels can be used to attack not only data that is currently stored in an encrypted table, but also deleted data that was formerly stored in that table. We also demonstrate how a number of algorithmic techniques not previously used in this space can dramatically speed up detection and extraction of secret data from widely used database storage engines. We then demonstrate that our attacks outperform the recent DBREACH compression side channel attacks (IEEE S&P 2023) in both functionality and performance. Specifically, we introduce a family of G-DBREACH attack techniques, each of which improves on one facet of prior work, and all of which can be used together in concert. First, the Ghost-DBREACH attack uses the template of DBREACH attacks, but also compromises deleted data from datatabases. Next, the Group-DBREACH attack uses group testing to accelerate string detection attacks. Finally, the Gallop-DBREACH attack shows how to replace various linear-cost operations in the previous attacks with logarithmic-cost ones, providing even more aggressive speedups of up to $9\times$.

Keywords: Compression, Side Channel Attacks, DBREACH

1 Introduction

Modern databases store immense quantities of data and back almost all critical applications on the web. To help affordably store these vast amounts of data on disk, database management systems (DBMS) offer compression features to reduce storage costs. Moreover, to protect users and customers' sensitive data, DBMSs support encryption of data at rest when it is written to disk.

Unfortunately, the combination of compression and encryption when dealing with sensitive data is a dangerous one [18]. Semantic security for encryption guarantees that a ciphertext hides everything about a message *except its length* [13],

but since compression reduces the size of a message in a content-dependent way – messages with more redundancy can be compressed more – compressing a message before encrypting it means that the message length will contain information that at least partially encodes the message content as well [18]. This vulnerability has been exploited repeatedly over the years (e.g., [6, 11, 17, 22, 26]), most famously in the CRIME and BREACH attacks on TLS [11, 22].

At a high level, compression side channel attacks take advantage of the ability of an attacker to place attacker-chosen strings next to victim secrets, such that the attacker and victim content are compressed and encrypted together. Observe that if the attacker manages to guess the victim secret exactly, the attacker and victim strings will compress, resulting in a shorter ciphertext. If the attacker string is unrelated to the victim secret, they will not compress, and the ciphertext will be longer. Thus, the basic prerequisites of a compression side channel attack are the attacker's ability to inject guesses next to victim data and the attacker's ability to observe the size of the resulting compressed and encrypted ciphertext. In the setting of communication over a network, attackers can take advantage of various web vulnerabilities to inject their guesses of user secrets, and the resulting packets sent over the network directly reveal their size.

Recently, DBREACH attacks have shown how to apply this general framework for compression side channel attacks to the database setting [14]. Database tables regularly store data for different users in different rows of the same table, opening a myriad of opportunities for an attacker to inject guesses at the secrets of other users, e.g., via unprivileged SQL commands or just modifying their user information over a web interface. Once the guesses are compressed, encrypted, and written to disk next to other users' data, an attacker with limited access to the server hosting the database can observe file sizes. Thus DBREACH attacks allow for a form of privilege escalation, where an attacker with limited access to the victim server can read database contents that it otherwise lacks permission to read.

Importantly, the database setting presents challenges not present when reading packet sizes over a network. Database storage engines and file systems allocate space in units of pages, not individual bytes, so a change of a single byte in a filesystem often results in no discernible change in file size. DBREACH attacks need to align the contents of database tables of file system page boundaries to ensure they can get meaningful information when compressed table sizes change. This requires a number of table updates for each injected guess to ensure that the guess is correctly aligned and to observe exactly how much compression occurs as a result of the guess.

1.1 Our Contributions

This paper introduces three new compression side channel attacks, which we collectively term G-DBREACH attacks (Ghost-DBREACH, Group-DBREACH, Gallop-DBREACH), as well as defenses to mitigate them. G-DBREACH attacks broaden the scope of known compression side channel attacks on databases and reduce the number of table updates required to run an attack. Since the

attack time depends entirely on the required number of table updates, these improvements make compression side channel attacks on databases more practical and better demonstrate the risks posed by this class of attacks. The improvements in the three G-DBREACH attacks operate independently of each other, improving different aspects of the attack process, and can be used together at the same time.

Ghost-DBREACH. We begin by making the observation that compression side channels expose not only data that is currently stored in a compressed and encrypted database table, but also data that was once stored in a table but has since been deleted. We find that we can attack recently deleted data in a table with almost the same effectiveness as data currently stored in the table. We are able to do this because compression side channels arise from the on-disk representation of a table, not the logical representation maintained by a database storage engine.

In many storage engines, deleted data is not truly removed and overwritten when a user deletes a row. Instead, a flag is set to indicate that the row is deleted, and subsequent SQL queries skip over deleted rows. The deleted space may be reused for other rows of similar size in the future, but until this happens, the data that has been 'deleted' remains on disk. Here, it is inaccessible via normal queries to the database, but our attacks circumvent the normal interface and directly interact with the data actually stored on disk. Thus we can detect and extract sensitive strings stored in a table, even after an application user or database owner has deleted them through the database's standard SQL interface.

Group-DBREACH. Previous compression side channel attacks, both on databases and in other contexts, focus on testing a single guess string at a time to see how well it compresses with existing table contents. However, we observe that most guesses being tested do not appear in the victim data being targeted by an attacker. In the database setting, we can eliminate multiple potential guesses at once by concatenating the guesses and inserting them into the table together. Instead of looking for a yes/no answer to whether a string is in the table, we seek a no/maybe answer. That is, we want to know if all of the included guesses do not appear in the table, or if it is possible that one of them does appear in the table. In the case that one of the included guesses does appear in the table, we can re-test that guess individually in the conventional way. As long as most guesses do not appear in the table, this approach can significantly reduce the number of table updates needed to test a given set of guesses. Realizing this approach in a concrete attack requires changes to how the attacker computes and assesses compressibility scores, as well as significant empirical testing to determine the parameter regime where group testing results in performance improvements. Our evaluation shows that this kind of grouping does result in improved performance, despite the inherent noisiness of assessing compressibility for a batch of messages at once. We believe that the general approach explored by Group-DBREACH attacks when attacking databases can also be extended to compression side channel attacks in other contexts.

Gallop-DBREACH. Finally, we show how to speed up Ghost-DBREACH and Group-DBREACH attacks so they require fewer database queries to run by introducing Gallop-DBREACH. Although there are different kinds of attacks proposed in prior work – detecting whether a given string is in a table, ranking the strings most likely to appear in a table out of a given set, and extracting secret strings from a table character by character – all three attacks rely on a *compressibility scoring* mechanism that relies on counting how many bytes of incompressible text one would need to add to a table before it overflows to a new page [14]. The heart of the compressibility scoring algorithm involves inserting a guess of a string in the table and then repeatedly updating the table to add or remove incompressible filler data (depending on the targeted storage engine) until the physical space used on disk changes. We accelerate the compressibility scoring process by using binary searches to determine the physical page boundary, rather than changing the size of the filler data one byte at a time until a change is observed.

Our Gallop-DBREACH speedups, when run on MariaDB, enable new kinds of attacks that would not be possible with the original DBREACH attacks. For example, in the best-case configuration where a database query requires about 1 second to flush to disk, the DBREACH attack benchmarks all complete in under an hour using Gallop-DBREACH, whereas they take 2-7 hours with the original attack. Since DBREACH and G-DBREACH attacks run on databases whose actual contents (victim data) are static during the attack, this means that G-DBREACH can target databases that get regular updates for a large section of a day but which occasionally have lulls in activity for a while. Examples of this kind include a database of prescription filling data at a pharmacy that closes for lunch, or a database of harassment/discrimination reports at a university where there are a few early morning hours with little activity.

We demonstrate our attacks against the InnoDB storage engine in MariaDB and the WiredTiger storage engine in MongoDB, comparing both accuracy and the number of table updates (a less noisy proxy for wall clock attack time) to DBREACH attacks. Overall, our improvements have minimal impact on attack accuracy while reducing attack time, sometimes significantly. Using the same database setup, compression algorithms, and test data sets as DBREACH, Group-DBREACH experiences speedups of up to 3× compared to DBREACH, and Gallop-DBREACH runs up to 6× or 9× faster than DBREACH on MongoDB and MariaDB, respectively. See Section 6 for evaluation details. Our attack demonstration code and raw evaluation data are publicly available at `https://github.com/bnbourassa/gdbreach-attacks`.

In addition to describing G-DBREACH attacks and measuring their performance, we discuss approaches to mitigating them in Section 7. Beyond the known mitigations for DBREACH, we make some new suggestions for developers and discuss additional considerations and mitigations that only apply to the new Ghost-DBREACH attack.

2 Background

This section provides the necessary database and compression background for our attacks. It also includes a summary of the relevant aspects of DBREACH attacks, over which our contributions improve.

2.1 Database Background

A storage engine is the component of a database management system that manages maintaining and updating the content of tables, as well as their storage format on disk. To demonstrate our techniques, we focus on two storage engines: InnoDB and WiredTiger. InnoDB is the default storage engine for MariaDB [1], and a similar storage engine is the default for MySQL [3]. WiredTiger [20] is the default storage engine for MongoDB [2]. These storage engines take dramatically different approaches to storing and organizing data, providing diverse contexts for testing the practical effectiveness of attacks.

We focus on the common core aspects of storage engines that enable our attacks, ignoring important elements like table metadata, keys, and indexes that will not be relevant to our discussion. Throughout the paper, we will use the language of relational database (e.g., tables, rows, columns) when describing both storage engines for consistency, although this is not always the terminology used in the documentation of platforms we examine.

Storage behavior. Both storage engines store rows in continuous sections of a file, with row header information preceding each row. Table files start small and grow in increments of a fixed page size when they exceed their allocated space (16kb pages in InnoDB, 4kb for WiredTiger). In general, new rows are stored contiguously with old ones, and updates to a row overwrite the previous contents, space permitting. When a row is deleted, the row is marked as such in the table file, but there is no other change to the data stored on disk. Subsequent insertions can overwrite the deleted data if they fit in the space formerly occupied by the deleted row. There are exceptions and special cases that apply to these general rules, but we have empirically found this behavior to be sufficiently consistent to enable our attacks.

Compression. We demonstrate our algorithmic improvements to the DBREACH attacks against three compression algorithms on InnoDB – zlib [10], LZ4 [28], and Snappy [4] – and against two compression algorithms on WiredTiger – zlib and Snappy. Although the details of the compression algorithms will not be critical to the exposition of our attacks, we briefly compare them for completeness.

The zlib, LZ4 and Snappy algorithms are all based on the LZ77 compression algorithm [29]. The zlib algorithm uses the `DEFLATE` file format and is the most aggressive of the three algorithms. It combines LZ77 [29] with Huffman coding [15]. The Huffman coding step introduces additional noise into the compression side channel and makes the original DBREACH attacks more challenging to perform against the zlib algorithm. LZ4 [28] is less aggressive and uses only LZ77 and no

Huffman coding. Snappy [4] is an open source compression library from Google that is based on LZ77, and is the least aggressive.

More important than the details of the compression algorithms themselves are the details of how the storage engines use these algorithms. InnoDB compression takes place within the context of a single database page, i.e., compression occurs separately in each 16KB chunk of the table. Since the underlying file system pages are generally 4KB large, compression allows the file system to reclaim extra space via a technique called *hole punching* [7]. Hole punching allows the file system to reclaim extra 4KB file system pages inside of each 16KB database page. For example, if a table is only using 10KB of a 16KB page, the file system can reclaim the unused 4KB page.

WiredTiger frees 4KB pages back to the file system when compression shrinks a table to the point where they are no longer needed, although the details of how this is done depends on the chosen compression method.

Encryption at Rest. While database tables are generally not encrypted in memory when they are accessed and subject to queries, many storage engines support encrypting tables at rest whenever they are stored to disk. Encryption at rest prevents attackers with access to the filesystem from accessing the tables. This is exactly the threat model for our attacker, and encryption prior to writing to disk is the safeguard that DBREACH attacks circumvent.

Both InnoDB and WiredTiger use AES in CBC mode to encrypt files to disk, although they support a variety of other modes and various key lengths. Ultimately, the choice of encryption algorithm and key size make no difference for our purposes. The only aspect of the ciphertext that we use is its length. The important design point that we leverage for our attacks is that encryption happens immediately before writing to disk and after any compression has occurred.

2.2 DBREACH Attacks

DBREACH attacks allow an attacker to detect or extract plaintext secrets from database tables that are compressed and encrypted on disk [14]. This section summarizes key elements of DBREACH that will be relevant to understanding our improvements.

Threat model. The attacker needs two capabilities in order to launch a DBREACH attack:

1. The ability to insert or update rows within the same compression window as the victim data.
2. The ability to measure the size of compressed and encrypted tables on disk.

Insertion and updating of rows in tables are common backend operations of user-facing web applications, so a myriad of applications give potential attackers the first capability. It is important, however, that attacker-controlled data be placed close to victim data. Attackers can achieve this by ensuring that they have inserted lots of data into tables over time, or by focusing on attacking data recently inserted by other users.

The requirement that attacker-controlled data be placed close to victim data means that DBREACH attacks are better suited to cases where victim data consists of short secret strings rather than large files. For example, a large data blob corresponding to the contents of a video or audio file may be so large that it fully contains the compression window within the file. In this case, there is no chance for adversary-controlled data to compress with certain parts of the victim file. Usernames, passwords, credit card numbers, social security numbers, etc., on the other hand, are small enough that even a relatively large record of identifying user data is small enough to fit many rows inside the same compression window.

Another opportunity to give an attacker the capability to insert data alongside honest users arises because database permissions can be configured to allow users to access or modify certain columns of a table but not others. This suffices to give the attacker the first condition of a DBREACH attack, as the attacker can modify any row near the row where it wants to extract the contents of the column it can't access. The DBREACH paper demonstrates insertion of data both through direct SQL queries and through abuse of a web page interface.

Accessing the size of tables when they are stored on disk is a more challenging condition to meet. To access this information, an attacker needs to have access to the file system where the table files are stored. Thus DBREACH attacks can be thought of as privilege escalation attacks. An attacker who has access to the filesystem storing sensitive data (and can therefore read file sizes), but who does not have root access to the database or access to the storage encryption keys, can use DBREACH to recover data stored in the database that it otherwise does not have permission to see.

Overview of attack types. DBREACH attacks are divided into three categories:

1. *Decision attack*: determines whether a given string is in the table.
2. *k-of-n inclusion attack*: determines, out of a list of n strings, which k are most likely to be in the table.
3. *Character-by-character extraction attack*: extracts a whole string from the table without prior assumptions about the table's contents other than a known prefix or suffix in the table.

All three attacks make use of the same technique (which our work improves), but use it in different ways. At the heart of the attack is an algorithm for determining, given an attacker's guess at a string in the table, how much the attacker's guess compresses with text already present in the table. This algorithm produces a *compressibility score*, which is used in different ways depending on the attack.

In the decision attack, a decision threshold is determined empirically based on characteristics of the expected data set (e.g., names, emails, etc.), and strings that score over the threshold are determined to be in the table, whereas strings below the threshold are not. Appendix A briefly describes how thresholds are computed in the original DBREACH paper. The k-of-n attack does not need thresholds because it can simply sort the strings by compressibility score.

Character by character extraction proceeds similarly to the k-of-n attack by making the observation that there are a fixed number of possible next characters

in a string being extracted. For example, in a string of lowercase letters, there are only 26 possible next characters, and in a string of numbers, there are only 10. Given a known prefix or suffix of a target string to be extracted (e.g., "`@gmail.com`"), an attacker can try all the possible strings that add a single character to the prefix/suffix and select the one most likely to be in the table. That is, the attacker tries "`a@gmail.com`," "`b@gmail.com`," ..., "`z@gmail.com`," to get the last letter of a targeted email address, repeats this process for the penultimate letter, and so on. DBREACH attacks run this step repeatedly in a series of amplification rounds to build confidence in the most likely character.

Measuring compressibility. The key step in any DBREACH attack, regardless of attack type, is repeatedly measuring the compressibility of various guess strings. This makes up the overwhelming majority of the time taken in the attack, so speeding up this step improves every part of attack performance. We now briefly summarize the DBREACH approach to compressibility scoring.

A one-time setup step begins by aligning the contents of a table with a filesystem page boundary. That is, making the contents of the table slightly more or less compressible will change the table size. Whether the table is just large enough that a little more compression will shrink it or just small enough that a little less compression will grow it depends on the storage engine being attacked (InnoDB and WiredTiger, respectively), but the attack is intuitively the same. We will describe the former case. In either case, the table is filled to this page boundary using random, incompressible filler data.

For each guess it wants to evaluate, the attacker updates a row it controls to contain the guess. Then it replaces the incompressible filler data from the attack setup with a very compressible string, one character at a time. For example, it could replace the filler data with the string "`aaaaaa`" character by character. After each insertion, it measures the size of the table, recording the number of bytes that needed to be changed before the table size shrinks. For a guess g, this count is referred to as c_g. Intuitively, this value is lower if g is present in the table and compresses with another occurrence of g, and larger if g is not present in the table and cannot compress. The filler state is reset in between guesses with another table update.

Unfortunately, while the value of c_g is correlated with how compressible a guess is, setting a threshold for c_g to use as a decision criteria for whether a string is in a table fails to provide good results. This is because c_g can be affected by other table contents, making it a noisy measurement. To get a more reliable value, the attacker also computes reference values c_{yes} and c_{no}, which are the counts for strings which the attacker knows are and are not in the table, respectively. The c_{yes} string is a substring of the attacker-inserted filler data, and the c_{no} string is a random string, which is highly unlikely to appear in the table. For each guess g, the attacker uses these scores to compute a decision value

$$d_g = \frac{c_{no} - c_g}{c_{no} - c_{yes}}.$$

The same *yes* and *no* strings can be used for all guesses, but the *yes* and *no* strings are the same length as the guess, meaning different reference values c_{yes} and c_{no} must be calculated for each guess length. This means that, for a guess set G, the attacker needs to compute $|G| + 2|L|$ scores in order to complete its attack, where $|L|$ is the number of distinct string lengths in G. Computing these scores accounts for almost the entire time required to run the attack.

3 Ghost-DBREACH: Attacking Deleted Data

We now describe how compression side channel attacks can be used to extract data that was once stored in a compressed and encrypted table, but which has since been deleted. This section describes how our targeted storage engines handle physical storage and removal of deleted data before describing considerations involved in extracting deleted data.

3.1 Physical Storage of Deleted Data

As mentioned in Section 2.1, neither InnoDB nor WiredTiger actually deletes or overwrites data that has been logically deleted from a table. Instead, they mark a row as deleted, which causes further selection queries to skip it and enables table updates to overwrite its contents. Newly inserted data can overwrite the contents of a deleted row only if it fits inside the gap left by the removed data, otherwise it would corrupt the contents of adjacent, non-deleted rows.

InnoDB and WiredTiger do provide ways for database administrators to truly remove deleted data from tables to save space on disk, but these operations require administrative commands distinct from the `DELETE` query called by applications to remove rows. For example, MariaDB offers the `OPTIMIZE TABLE` command to optimize the contents of a table file, and WiredTiger uses the `compact` command for a similar purpose. While these operations may be run occasionally to optimize file storage, we do not expect applications to run them regularly enough to interrupt the execution of compression side channel attacks to extract recently deleted data.

3.2 DBREACH on Deleted Data

Our key observation is that the execution of compression side channel attacks on databases only relies on the *physical* state of a table file, not on the *logical* state of the table it represents. This means that the effectiveness of an attack should not be affected by whether a flag in the table file says a row is deleted, so long as the actual data for that row remains intact. As such, the same techniques used in DBREACH to attack the current contents of a table can be used in Ghost-DBREACH to attack the former contents of a table. However, we need to be careful lest the modifications made to a table during an attack should themselves overwrite the deleted data we seek to attack. We nonetheless hypothesize that DBREACH attacks are sufficiently robust to attack deleted data.

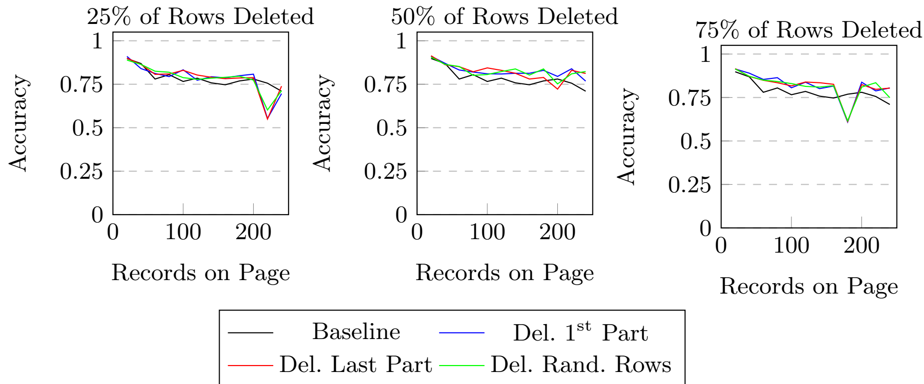

Fig. 1: Accuracy of the decision attack on the English dataset under the zlib compression algorithm when 25%, 50%, and 75% of the target data has been deleted from the table. Deleting data does not noticeably reduce attack accuracy.

To test this hypothesis, we re-ran a subset of the DBREACH decision attack experiments with segments of the table data deleted before running the attack. We deleted 25%, 50%, and 75% of the table data. To ensure that the pattern of deleted data does not matter, we tried deleting the first part of a table, the last part of a table, or random rows within a table. We carried out the attack against the InnoDB storage engine with zlib compression. The target dataset is a list of English words, and we varied the number of records stored in the table from 20 to 240. See Section 6 for more details of our experimental setup.

The results of our experiment on deleted data appear in Figure 1. The accuracy in each case remains similar to the accuracy achieved when the data has not been deleted, regardless of how much data is deleted or how the deleted data is situated within the table. We infer that since the initial filler rows inserted during the attack are longer than any row that has been deleted, the attack does not overwrite the deleted data still held in the table file. Inserting and deleting very long pieces of data may result in different outcomes, but our attacks are designed to run in the setting where multiple rows compress together within a single database page anyway, limiting the maximum relevant size for a single piece of targeted data.

4 Group-DBREACH: Group Testing for Faster Decisions

This section describes Group-DBREACH, the second of the G-DBREACH attacks. While the Ghost-DBREACH attack primarily capitalizes on an observation that expands the scope of prior work, our remaining attacks make algorithmic improvements to attack techniques to speed up the attack process. These improvements accelerate the speed of the attacks, reducing the overall time required without

reducing accuracy. This section and the following one describe Group-DBREACH and Gallop-DBREACH attacks, respectively, and the accuracy and speedups provided by the attacks are measured in Section 6.

4.1 Testing for the Tails

The original DBREACH decision attack was evaluated by trying to determine which of a long list of potential guesses was included in a table. The original evaluation had 50% of guesses included in the table.

We observe that there are many cases where an attacker has a long list of potential guesses and only expects a small fraction of them to be included in the table. For example, an attacker trying to find out which of a large number of names or email addresses appears in the members' list of a forum dedicated to discussing politically sensitive matters.

In cases like this, we show that it is possible to reduce attack time by adopting a group testing approach. Instead of checking whether each individual guess is or is not in the victim table, we instead test whether a group of guesses may have a constituent element that appears in the table. In the case where none of a group of guesses appear in the table, we can skip the whole group with a single compressibility score measurement, and otherwise we re-test each member of the group individually. For cases where we expect very few guesses to appear in the table, this approach can provide considerable savings. However, adopting a naïve approach to group testing can actually degrade performance. We now describe how our attacks approach group testing, identify the performance pitfalls of a naïve approach, and show how we overcome them.

4.2 Testing Groups

We implement this group testing idea by concatenating multiple guesses together into one *super-guess*. We composed super-guesses as groups of two individual guesses to test the simplest version of this idea, but we can increase group sizes by recalibrating the attack for larger numbers of guesses. However, the size of a file system page on disk does constitute a hard upper limit on the number of guesses per super-guess, and the more of a page is used up in the attack, the smaller the amount of data that can be attacked on that page.

The Group-DBREACH attack works in two passes. In the first pass, it computes reference scores and compressibility scores for super-guesses and filters out any guesses whose compressibility scores fall below the pre-computed decision threshold. The second pass repeats the process of computing reference scores and compressibility scores, but it excludes any guesses that belonged to super-guesses whose compressibility scores fell below the threshold in the first pass.

4.3 Reference Scores for Group Testing

The attack as described thus far reduces the number of guesses that need to be tested, but it often actually *increases* the number of table updates required to

complete the attack. To see why, recall that DBREACH attacks compute separate reference scores c_{yes} and c_{no} for each guess length. Concatenating multiple guesses of different lengths can increase the number of distinct guess lengths being tested, thereby increasing the number of reference scores needed. We find this to be the case for the test data sets used to evaluate DBREACH. Although group testing reduces the overall number of guesses/super-guesses required, the computation of additional reference scores cancels out these improvements.

Histograms of String Length Distribution

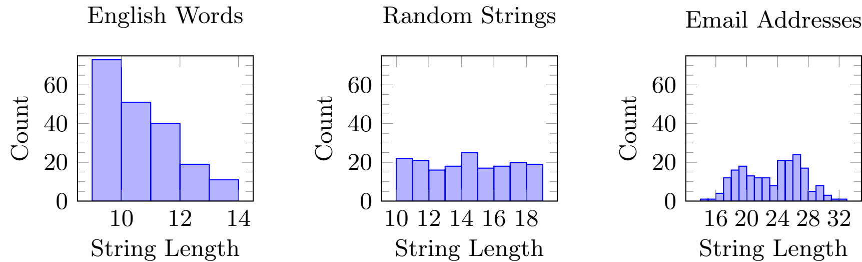

Fig. 2: Distribution of string lengths in our evaluation data sets.

Reference Scores for Each Dataset

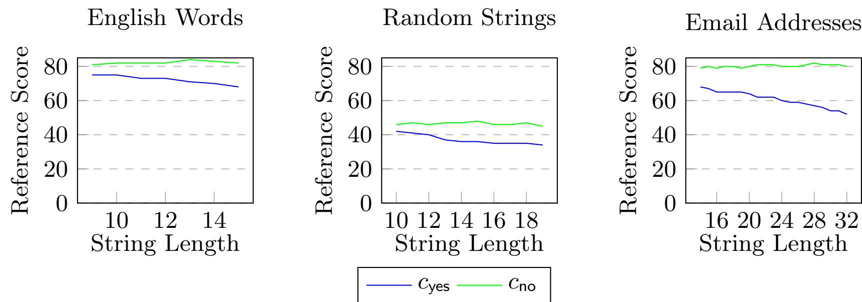

Fig. 3: Reference score distribution for each of our evaluation data sets. The x-axes are aligned with the range of string lengths that occur in each data set. Values of c_{no} remain relatively constant as string lengths increase, and values of c_{yes} gradually decrease.

To reduce the overhead incurred by computing reference scores, we adopt a more intelligent approach to reference score calculation. Our approach is based on two observations:

1. Since the attacker always knows its list of guesses in advance (this is true for both detection and character-by-character extraction attacks), it also knows what reference scores need to be calculated before starting the attack. For example, Figure 2 shows the lengths of strings included in the various data sets used in our attack evaluation.

2. When using reference scores to calculate compressibility scores, the important quantities are the value of c_{no} and the difference $c_{no} - c_{yes}$. While there is some variation in reference scores for shorter and longer strings, changing the length of a guess by one byte has little impact on the reference score. In particular, c_{no} remains largely constant and c_{yes} decreases gradually as the length of the reference string increases. This is illustrated by Figure 3, which shows the average values over ten trials of c_{yes} and c_{no} (using the zlib compression algorithm) across each data set used in our evaluation.

   This behavior is to be expected because c_{no} is computed by replacing incompressible bytes with different incompressible bytes and measuring how many of these need to be made compressible for the table size to change. This has nothing to do with the length of the guess. On the other hand, measuring c_{yes} involves replacing incompressible bytes with a string that is known to appear in the table and which must therefore compress very well. This can be thought of as giving the process of replacing incompressible bytes with a compressible ones a "head start" that increases in length as the reference string length grows.

Based on these observations, our attack calculates all the reference scores for super-guesses before starting the group testing process, and then calculates a new set of reference scores for individual guesses before evaluating those guesses. In both sets of reference scores, it only calculates the reference score for every seventh length, a value we determined empirically to balance accuracy and performance. Compressibility scores are calculated using reference scores for the nearest length available. Using this *approximate reference score* strategy significantly reduces the number of reference scores that need to be calculated, and eliminates the additional reference score overhead introduced by naïve group testing.

We made one final optimization to unlock the performance improvements possible with Group-DBREACH attacks. Recall that when measuring the compressibility of a string, DBREACH attacks start by writing the string over a series of incompressible bytes. Since forming super-guesses by concatenating several strings results in extra-long strings, there is a chance that a super-guess has enough compressibility with other strings in the table that the table size changes before the attack can begin the process of replacing additional incompressible bytes with compressible ones. DBREACH attacks handle this situation by giving up and restarting the attack. If Group-DBREACH exhibited the same behavior, the slowdowns from this increasingly frequent event would once again negate all performance improvements. Instead, we observe that the c_{yes} reference score in these cases is always close to 1, meaning that we would expect a string of this length in the table to very quickly result in a table size change anyway. Therefore, instead of resetting in the event of premature table size changes, we always assign $c_g = 1$ and continue the attack. This changes a situation that could reduce attack speed into one where we can take a shortcut to further speed up the process. As demonstrated by our evaluation in Section 6, this shortcut does not result in any loss of attack accuracy.

5 Gallop-DBREACH: Speedy Compression Scoring

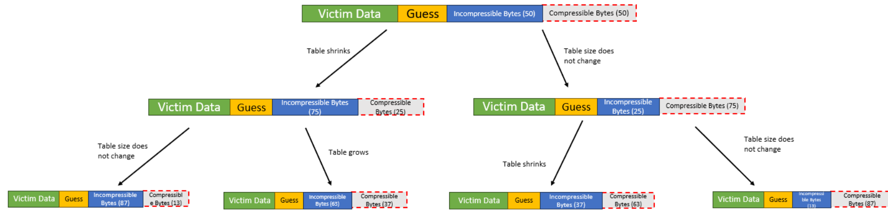

Fig. 4: An example of how the Gallop-DBREACH binary search for calculating compressibility scores works. The attacker begins by inserting a guess, $B/2$ incompressible bytes, and $B/2$ compressible bytes. If the table shrinks, the attacker updates the table to increase the number of incompressible bytes and lower then number of compressible bytes. If the table does not shrink, the attacker lowers the number of incompressible bytes and increases the number of compressible bytes. This process continues until the attacker discovers the minimum number of bytes that must be changed to alter the size of the table. If the binary search ends up inserting B compressible bytes into the table without a size change, we repeat the binary search again between B and $2B$ bytes.

This section presents Gallop-DBREACH, the last of the G-DBREACH attacks. We chose this name because the attack causes the table size to rapidly rise and fall, like the hooves of a galloping horse that carries us to faster compressibility score results.

5.1 Linear to Logarithmic

Recall that the heart of a DBREACH attack consists of repeatedly updating a table until the size on disk changes, and using the number of updates to compute a compressibility score. The first update inserts the attacker's guess string g into a table near the victim row followed by a number of incompressible filler characters, and each subsequent table update replaces an incompressible character with a compressible one. This process gradually replaces a random string following the guess with a string composed entirely of repetitions of the same character (which will compress very well). The attacker's goal is to count the number of bytes swapped in the table before the size on disk changes. A larger number of bytes corresponds to a less compressible guess, and a smaller number of bytes corresponds to a more compressible guess because it suggests that the guess itself was compressing with content already in the table.

We observe that each update to the table rewrites the same section of the file holding the table, as all the updated strings are the same length, changing only the distribution of compressible and incompressible bytes in the string. Thus,

there is no reason to change bytes one at a time, except to precisely measure how many bytes need to be changed to shrink the table on disk.

Taking advantage of our observation, we replace the linear and incremental changes of the original DBREACH attack with a binary search. However, running a binary search requires a lower and upper bound to search between. The linear DBREACH attack starts at zero swapped bytes and increments this until the table size changes. Our binary search can't know how many bytes will need to change in the worst case a priori (unless it picks an unnecessarily large upper bound), so we conduct the binary search in a series of B-byte segments. In practice, we use $B = 100$. This change balances the performance benefits of binary search in the common case while maintaining the flexibility offered by not assuming a pre-determined upper bound on the number of bytes that may need to be changed.

Concretely, instead of changing one byte at a time, we begin by replacing the first $B/2$ incompressible bytes with compressible ones. If the table shrinks, we know we have overshot the required number of bytes and go back to only having $B/4$ compressible bytes in the next step. If the table does not shrink, we increase the number of compressible bytes to $3B/4$ in the next step, and so on. Figure 4 illustrates this process. If the number of compressible bytes reaches B, this means that the table doesn't shrink by changing B bytes. In this case, the process repeats with the next B bytes of incompressible filler, starting by replacing $B/2$ of the next B bytes with compressible bytes.

5.2 Storage Engine-Dependent Details

While Gallop-DBREACH operates the same in principle for all target storage engines, implementation details of the attack on different storage engines result in some changes.

- *MariaDB + InnoDB:* When attacking the InnoDB storage engine, the attacker repeats the same process when it computes the reference scores and when it measures the compressibility of guess strings. Thus we use the binary search in the same way when computing all three of c_{yes}, c_{no} and c_{g}.
- *MongoDB+WiredTiger:* In running the attack against the WiredTiger storage engine, the attack re-uses the c_{no} reference score when computing c_{yes} and c_{g}, starting the number of compressible bytes at the number used for c_{no}. This is possible because in MongoDB, our attack aims to add incompressible bytes until a table grows, rather than adding compressible ones until the table shrinks, as in MariaDB. This means that the cost of computing c_{no} is by far the most time-consuming when attacking this storage engine. Thus, we only implement the binary search when computing the c_{no} reference score. This means that speedups on MongoDB depend on the number of guesses being tested, as testing many guesses of the same length reuses the same c_{no}. Trying out a small number of guesses gets very large speedups, while larger numbers of guesses get smaller speedups. See Section 6 for details.

6 Evaluation

This section measures the effectiveness of our various attack techniques at speeding up compression side channel attacks on databases compared to DBREACH attacks. We focus on the Group-DBREACH and Gallop-DBREACH attacks, as the accuracy and performance of Ghost-DBREACH is identical to standard DBREACH, although Ghost-DBREACH can also be sped up using the techniques of the other G-DBREACH attacks.

Attack environment and parameters. We ran our experiments on a Google Cloud, GCP, e2-medium instance, with an Intel® Broadwell 2.20GHz CPU with 2 cores and 4 GB RAM. The instance uses SSD storage. We installed MariaDB 10.3.37 and MongoDB Community Edition Server on top of Ubuntu 20.10.

We measure the running time of our attacks in terms of the number of table updates required to launch the attack. This serves as a less noisy and more system-independent proxy for the wall clock time of the attack. As in the original DBREACH evaluation, we configure the database to flush pages to disk after each insert or update in order to more quickly measure the accuracy and number of updates required. In this configuration, attacks take a few seconds per guess, and often take as little as a second per guess on MariaDB. Numbers reported here are the averages of 10 or 50 trials, as specified for each portion of the evaluation.

We test the G-DBREACH and DBREACH attacks when the victim database is configured to use the zlib, LZ4, and snappy compression algorithms, demonstrating that G-DBREACH attacks are effective against a broad spectrum of compression algorithm choices.

To ensure a fair comparison, we evaluated our attacks using the same three data sets used in the evaluation of DBREACH [14], to which we compare our results. The three data sets are composed of random strings, English words, and email addresses, respectively. The scheme for each table attacked consists of `id` and `value` columns, where `id` is an integer and `value` is a string. The choice of which elements from the data set are included in the victim table and attacker's guess list is randomized for each attack, but we run the DBREACH and G-DBREACH attacks for each random choice to ensure differences do not arise from randomness in the attack setup.

When running attacks that involve the attacker making decisions about whether or not a guess string appears in a victim table, we have the attacker decide based on whether or not the compressibility score for the guess exceeds a given threshold. We use the thresholds originally computed for use in DBREACH attacks, which appear in Appendix A. Note that the choice of threshold makes no difference for the time required to compute a compressibility score, but it may make a difference in the Group-DBREACH attack if an inaccurate threshold results in too many groups testing positive as potentially appearing in the table. It is thus possible that our results may be improved by calculating distinct thresholds for grouped data.

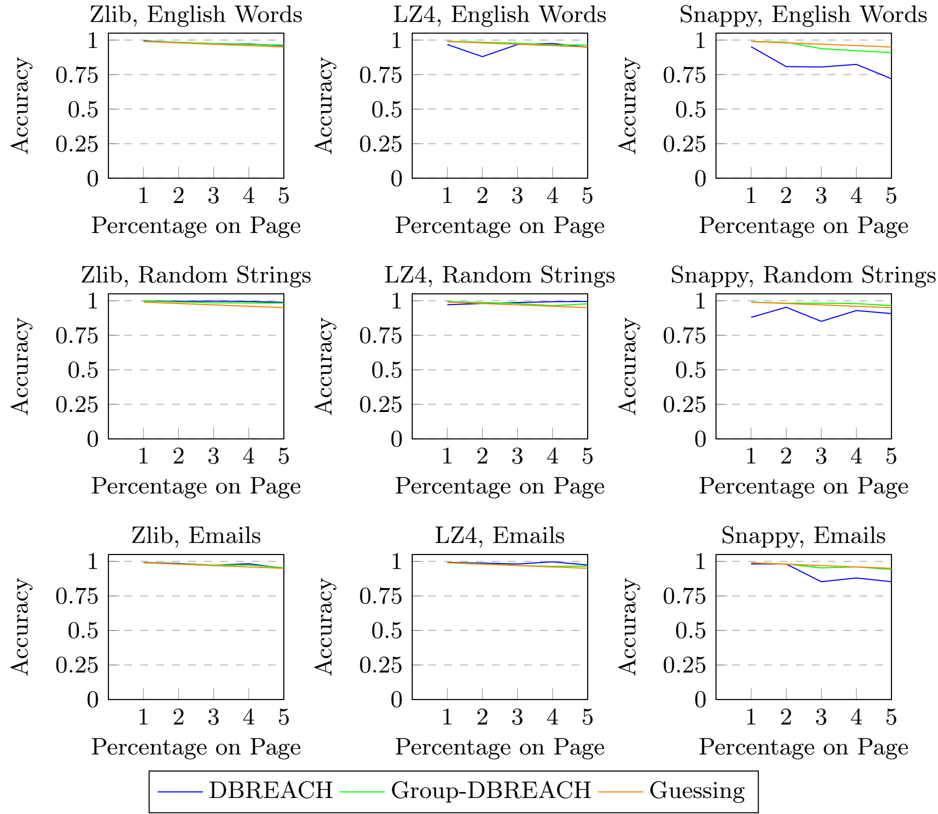

Fig. 5: Comparison of Group-DBREACH accuracy with DBREACH and guessing by tossing a weighted coin. Group-DBREACH always matches or exceeds the accuracy of DBREACH and almost always surpasses guessing by weighted coin, a strategy that is much more likely to generate false positives. Numbers are averages of 50 trials.

6.1 Group-DBREACH

We tested the Group-DBREACH attack, described in Section 4, against the InnoDB storage Engine. Our experiment consists of a victim table of 100 rows whose entries are elements of one of our test data sets. The attacker is given a list of 100 guess strings, of which 1-5% are included in the victim table, and it must decide whether each guess does or does not appear in the table. This is a much more challenging evaluation setting than the one used for the original evaluation of DBREACH. Numbers reported for the Group-DBREACH evaluation are the average of 50 repeated trials.

Accuracy. Figure 5 measures the accuracy of Group-DBREACH attacks, comparing the accuracy of our attack with that of either DBREACH or random guessing. The random guessing approach consists of tossing a weighted coin, which decides

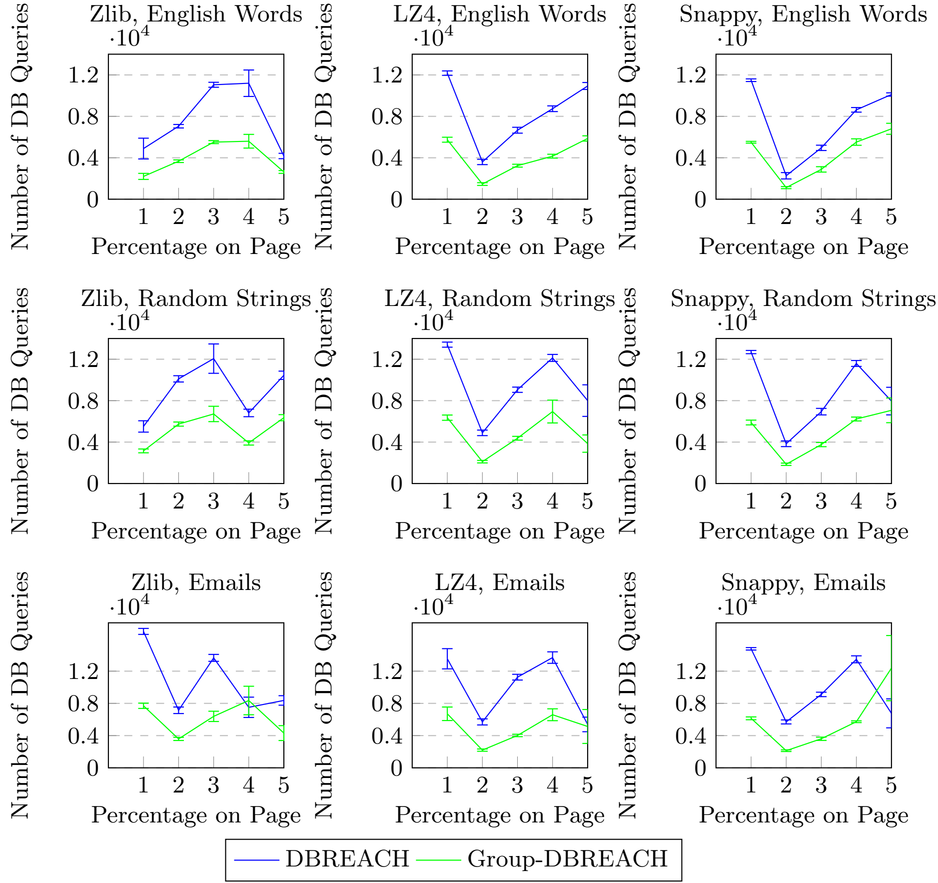

Fig. 6: Comparison of number of table updates when running Group-DBREACH and DBREACH attacks. Group-DBREACH consistently reduces attack time when small fractions of guesses appear in the victim table, resulting in speedups of up to 2.8× and an average of 2× speedup on average across all tested scenarios. Numbers are averages of 50 trials.

a string is in the table with probability proportional to the likelihood that it is actually in the table. This results in accuracy of 99-95% as the percentage of the attacker's guesses that appear in the table increases from 1%-5%, but it does not represent a meaningful real attack. It both presupposes knowledge of exactly how many guesses will be in the table and is extremely prone to false positives. Nonetheless, we include it as a naïve baseline benchmark that does well in this evaluation setup. Group-DBREACH matches or exceeds the accuracy of the two baselines in all cases.

Speedup. Figure 6 shows the number of database updates needed on average to complete the attack. These results varied based on the data set and compression algorithm used by the database. In all cases, our grouping technique resulted in improvements up to at least 4% of the attacker's guesses being included in the table, and in most cases the improvements continued past the 5% threshold where we stopped testing. On average across all our tests, we are able to speed up the attack by 2.0×, and some cases see improvements of up to 2.8×. This experiment demonstrates that group testing results in clear attack speedups even when using the smallest group size possible – 2 elements in a group.

	DBREACH		Gallop-DBREACH		
Case	Accuracy	DB Queries	Accuracy	DB Queries	Speed Up
zlib, English	0.79	10,775.3	0.77	2,074.5	5.2x
LZ4, English	0.906	14,859.9	0.825	2,073.7	7.2x
Snappy, English	0.748	11,898.1	0.748	2,058	5.8x
zlib, Random	0.922	26,112.9	0.91	2,782.4	9.4x
LZ4, Random	0.991	21,076.5	0.964	3,105.2	6.8x
Snappy, Random	0.972	12,987	0.988	2,072.5	6.3x
zlib, Emails	0.918	13,512.2	0.87	2,336	5.8x
LZ4, Emails	0.94	7,845.8	0.914	2,995.7	2.6x
Snappy, Emails	0.85	20,397	0.812	2,489.1	8.2x

Fig. 7: Comparison of DBREACH and Gallop-DBREACH decisions attack accuracy and performance when run on MariaDB+InnoDB. Gallop-DBREACH consistently results in dramatic performance improvements at a very small cost in accuracy. Numbers are averages of 10 trials.

6.2 Gallop-DBREACH

We tested Gallop-DBREACH attack, described in Section 5, against the InnoDB and WiredTiger storage engines. For this attack, we tested the performance of the attack in decision attacks, as we have for prior attacks, as well as character-by-character extraction attacks, with the aim of demonstrating that the improvements we demonstrate in decision attacks also result in improvements in the more challenging string extraction setting.

For the decision attacks, our experiment consists of inserting 100 rows into the victim table and giving the attacker a list of 200 guess strings, of which 100 are in the table. To illustrate how the Gallop-DBREACH attack has a greater effect on MongoDB results when run with a smaller number of guesses, we also run our experiment with a modification where we insert 20 rows into the victim table and the attacker has a list of 40 guess strings to test. Both these attack scenarios are borrowed directly from the original DBREACH attack evaluation

		DBREACH		Gallop-DBREACH		
Case	Number of Guess Strings	Accuracy	DB Queries	Accuracy	DB Queries	Speed Up
zlib, English	20	0.992	5,018.8	0.988	853.5	5.8x
zlib, English	100	0.836	5,569.2	0.836	3,061.1	1.8x
Snappy, English	20	0.948	4,157.3	0.928	943.2	4.4x
Snappy, English	100	0.908	4,659.1	0.906	3,701.8	1.3x
zlib, Random	20	0.988	4,978.3	0.992	1,183.4	4.2x
zlib, Random	100	0.924	5,689.2	0.832	4,096.5	1.4x
Snappy, Random	20	0.992	4,249.3	1.0	1,291.7	3.3x
Snappy, Random	100	1.0	4,531.7	0.996	4,103	1.1x
zlib, Emails	20	0.968	5,874.3	0.985	2,468.6	2.4x
zlib, Emails	100	0.934	9,042.1	0.911	7,985.4	1.1x
Snappy, Emails	20	0.968	4,956.4	0.98	2,379.5	2.1x
Snappy, Emails	100	0.919	9,231	0.923	8,732.3	1.1x

Fig. 8: Comparison of DBREACH and Gallop-DBREACH decisions attack accuracy and performance when run on MongoDB+WiredTiger. Gallop-DBREACH results in significant performance improvements for smaller numbers of guesses, but improvements become more modest as the number of guesses increases. This is due to the MongoDB+WiredTiger version of the attack only affecting the computation of the c_{no} reference score, which is computed once for each guess string length and reused. Numbers are averages of 10 trials.

	DBREACH		Gallop-DBREACH		
Trials	Accuracy	DB Queries	Accuracy	DB Queries	Speed Up
50	1.0	74,144.02	1.0	46,863.08	1.6x

Fig. 9: Results for character-by-character extraction in MariaDB+InnoDB, including accuracy and number of database queries required for DBREACH and Gallop-DBREACH. Gallop-DBREACH speeds up the attack 1.6× with no compromise in accuracy. Numbers are averages of 50 trials.

to ensure a fair comparison. Numbers presented for these attacks are the average of 10 repeated trials.

MariaDB+InnoDB results. Figure 7 shows the accuracy and database query counts for running the Gallop-DBREACH attack, as well as the corresponding figures for a DBREACH attack. On average, across all our attack scenarios, Gallop-DBREACH provides a 6.3× speedup over DBREACH, with a speedup of at least 2.6× in all test cases. This demonstrates consistent performance improvements across various data sets and compression algorithms. Concretely, considering the best case of about 1 second per database query, the G-DBREACH

attacks we test all run in under an hour, whereas the original DBREACH attack takes 2-7 hours in the same setting.

MongoDB+WiredTiger. Figure 8 shows the results of running the Gallop-DBREACH attack on MongoDB. On average across all our tests, Gallop-DBREACH provides about a 1.3× speedup when there are 100 guesses to be made. When there are only 20 guesses, we achieve an average speedup of about 3.7×. The difference in speedup reflects the amount of time the calculation of c_{no} reference scores takes during the attack. Recall that the Gallop-DBREACH attack on MongoDB only speeds up the calculation of c_{no} reference scores. Thus, when there are fewer guesses, reference score calculations take up a larger fraction of the total attack time, so speeding them up results in larger performance gains.

Character-By-Character Extraction. Figure 9 shows the results of using Gallop-DBREACH to speed up the decision attack subroutines of the character-by-character extraction attack. Recall from Section 2 that the character-by-character account consists of repeated decision attacks to gradually build up a longer string from a known prefix or suffix. Our test uses a known prefix length of 15 and measures the effectiveness of Gallop-DBREACH at correctly determining the next character. We compared the average accuracy of Gallop-DBREACH and DBREACH over 50 trials and found that both attacks had 100% accuracy at determining the next character. However, Gallop-DBREACH was able to determine the correct next character 1.6× faster than DBREACH, saving tens of thousands of database queries in the process. This demonstrates that the G-DBREACH attack performance improvements also result in improvements for character-by-character string extraction attacks, which rely heavily on a decision attack subroutine.

7 Mitigation

Like all compression side channels, the guaranteed way to protect against G-DBREACH attacks is to stop using compression in the application. This removes all potential for G-DBREACH attacks, but it is often impractical, expensive, or otherwise undesirable to stop using compression for database tables. After all, compression is an important tool for handling large-scale data sets.

For applications where fully removing compression is undesirable or infeasible, we present a number of techniques to mitigate the potential for G-DBREACH attacks. Since the Group-DBREACH and Gallop-DBREACH attacks exploit the same vulnerability as the original DBREACH attacks, the mitigations for these two attacks are very similar. At a high level, DBREACH opportunities arise when data from multiple users is compressed together, so reducing instances where this happens reduces the risk of DBREACH attacks. The DBREACH paper suggests that this can be done by using compression mechanisms that compress the contents of rows individually, rather than in larger sliding windows, or by exposing "compression buckets" to the user of a DBMS to choose what sets of data can compress together. An alternative approach would be for application

developers to handle compression and encryption within their applications, where more context is known about what data can be compressed safely, instead of relying on the database to add this new functionality. The result is equivalent from a security perspective, but this allows developers to make changes on their own without storage engine modifications.

Unlike Group-DBREACH and Gallop-DBREACH, mitigating Ghost-DBREACH involves considerations unique to protecting deleted data. Like the other G-DBREACH attacks, Ghost-DBREACH can be mitigated by storage engine changes, e.g., by fully overwriting rows when they are deleted. However, compelling options exist for reducing the Ghost-DBREACH attack surface without major additional developer effort. First, we observe that using table optimization commands like `OPTIMIZE TABLE` in MariaDB and `compact` in MongoDB eliminates lingering deleted data. However, these commands are generally run occasionally to optimize file storage and are not typically run with the same frequency as the commands involved in everyday table operations. Optimizing storage for an entire table after every deletion is extremely wasteful, incurring computational costs in the cost of the entire table size to delete one row.

Instead, application developers seeking to reduce their exposure to DBREACH attacks can simply overwrite table data with random strings of the same length before deleting them. For the kinds of small rows that are most vulnerable to DBREACH attacks in the first place, we found empirically that this often results in the data being overwritten before it is marked as deleted, giving the developer an opportunity to overwrite the sensitive data before it is returned to the control of the storage engine. We remark that more advanced techniques for secure deletion are not necessarily needed here because we are not trying to remove every trace of the data from the physical storage media. It suffices to make sure that the view of the table file seen by the DBMS no longer holds the sensitive deleted data, even if the bits of the data would still be visible, e.g., to an attacker with physical access to the hard drive.

8 Related Work

Kelsey [18] initiated research on compression side-channel attacks by pointing out the potential to detect or extract plaintexts in settings where compression is combined with encryption. These attacks were demonstrated as threats to real TLS and HTTPS connections in the CRIME [22] and BREACH [11] attacks, as well as their subsequent improvements and refinements [6, 12, 17, 19, 26]. As a special case of compression, memory deduplication can also be used to compromise encryption [24] over a network.

Compression side channel attacks have also been demonstrated in a number of non-web contexts. Most relevant to our work, DBREACH attacks [14] exploit compression and encryption of database tables at rest to detect or extract sensitive plaintexts. Schwartzl et al. [23] demonstrate timing side-channel attacks on memory compression, including an attack on Postgres databases, albeit in a stronger attacker model than DBREACH. Other attacks show how to exploit

compression in various aspects of processor architecture to circumvent encryption [25, 27]. Fábrega et al. [8] generalize compression and other kinds of side channels when introducing "injection attacks." One of their attacks also makes use of a binary search technique, but they do this in a very different way than we do.

Although the vulnerablilities of combining compression and encryption are inherent to the goals of the two tools, a number of schemes have attempted to get the best possible compression without compromising encryption [5,16,21]. Others attempt to work around compression side channels by ensuring that sensitive data cannot be compressed with other data [16] or using static analysis to check for the presence of compression side channels [21]. Recently, Fleischhacker et al. [9] have studied how to safely compress encrypted data under constraints on the structure/content of the data.

9 Conclusion

We have introduced G-DBREACH attacks, a family of three techniques for exploiting compression side channels on databases that expand the range of applicability of recent DBREACH attacks [14] while also significantly improving attack performance. Our attacks expand the scope of vulnerable data to include data that has been deleted from victim tables (Ghost-DBREACH), and accelerate the pace at which side channel attacks can detect strings in a table or exfiltrate secrets (Group-DBREACH and Gallop-DBREACH). Our speedups are enabled by improvements to the attack's compressibility scoring algorithm, including making use of group testing and binary search techniques. These improvements show that compression side channel attacks on databases are less costly than initially reported, and future work will no doubt continue to show further improvements in both the scope and speed of this family of compression side channel attacks.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and feedback to improve this paper. This work was supported by a gift from Cisco.

References

1. MariaDB. `https://mariadb.com/`.
2. MongoDB. `https://www.mongodb.com/`.
3. MySQL. `https://www.mysql.com/`.
4. snappy. `https://google.github.io/snappy/`.
5. Janaka Alawatugoda, Douglas Stebila, and Colin Boyd. Protecting encrypted cookies from compression side- channel attacks. In *Financial Cryptography*, 2015.
6. Tal Be'ery and Amichai Shulman. A perfect crime? only time will tell. *Black Hat Europe*, 2013.

7. Jonathan Corbet. Punching holes in files. `https://lwn.net/Articles/415889/`, 2010.
8. Andrés Fábrega, Carolina Ortega Pérez, Armin Namavari, Ben Nassi, Rachit Agarwal, and Thomas Ristenpart. Injection attacks against end-to-end encrypted applications. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 82–82. IEEE Computer Society, 2023.
9. Nils Fleischhacker, Kasper Green Larsen, and Mark Simkin. How to compress encrypted data. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part I*, volume 14004 of *Lecture Notes in Computer Science*, pages 551–577. Springer, 2023.
10. Jean-Joup Gailly and Mark Adler. zlib. `https://zlib.net/`, 1995.
11. Yoel Gluck, Neal Harris, and Angelo Prado. BREACH: Reviving the CRIME attack. *Black Hat*, 2013.
12. Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. Request and conquer: Exposing cross-origin resource size. In *USENIX Security*, pages 447–462, Austin, TX, August 2016. USENIX Association.
13. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.
14. Mathew Hogan, Yan Michalevsky, and Saba Eskandarian. Dbreach: Stealing from databases using compression side channels. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.
15. David Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
16. Dimitris Karakostas, Aggelos Kiayias, Eva Sarafianou, and Dionysis Zindros. CTX: Eliminating BREACH with context hiding. *Black Hat Europe*, 2016.
17. Dimitris Karakostas and Dionysis Zindros. New developments on BREACH. *Real World Crypto*, 2016.
18. John Kelsey. Compression and information leakage of plaintext. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2002.
19. Dimitris Krakostas, Dionysis Zindros, Eva Sarafianou, and Dimitris Grigoriou. Rupture. `https://github.com/decrypto-org/rupture`.
20. MongoDB. The WiredTiger Storage Engine. `https://www.mongodb.com/docs/manual/core/wiredtiger/`.
21. Brandon Paulsen, Chungha Sung, Peter A. H. Peterson, and Chao Wang. Debreach: Mitigating compression side channels via static analysis and transformation. In *IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 899–911. IEEE, 2019.
22. Juliano Rizzo and Thai Duong. CRIME. *Ekoparty*, 2012.
23. Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical timing side-channel attacks on memory compression. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1186–1203. IEEE, 2023.
24. Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote memory-deduplication attacks. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

25. Po-An Tsai, Andres Sanchez, Christopher W Fletcher, and Daniel Sanchez. Safecracker: Leaking secrets through compressed caches. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1125–1140, 2020.
26. Mathy Vanhoef and Tom Van Goethem. HEIST: HTTP encrypted information can be stolen through TCP-windows. *Black Hat*, 2016.
27. Yingchen Wang, Riccardo Paccagnella, Zhao Gang, Willy R Vasquez, David Kohlbrenner, Hovav Shacham, and Christopher W Fletcher. GPU.zip: On the side-channel implications of hardware-based graphical data compression. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 84–84, 2024.
28. Yann Collet. LZ4. `https://lz4.github.io/lz4/`.
29. Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

A Decision Thresholds

This appendix briefly describes the thresholds used by the original DBREACH attacks, shown in Figure 10. These thresholds are used when determining, during a decision attack, whether or not a string is in the victim table. The threshold was found by testing thresholds over the range -0.5 to 1.5 in increments of 0.01 [14], and measuring the accuracy of an attack that used each threshold compressibility score for deciding whether or not a string appears in the victim table. The maximum accuracy achieved in each case was found under the thresholds displayed in Figure 10. To evaluate the effectiveness of G-DBREACH attacks in improving performance while maintaining attack accuracy compared to prior work, we used these same thresholds for each compression algorithm, storage engine, and data set combination.

Target Data	Random	English	Emails
Snappy (IDB)	.50	.62	.85
zlib (IDB)	.45	.69	.81
LZ4 (IDB)	.47	.55	.78
Snappy (WT)	.35	.39	.49
zlib (WT)	.46	.35	.52

Fig. 10: Compressibility score thresholds determined by the original DBREACH attacks for each compression algorithm, storage engine, and target data setting. IDB denotes InnoDB, whereas WT denotes WiredTiger.