DBREACH: Stealing from Databases Using Compression Side Channels

Mathew Hogan
Stanford University
mhogan1@cs.stanford.edu

Yan Michalevsky
Anjuna Security, Inc.
and *Cryptosat, Inc.*
yanm2@cs.stanford.edu

Saba Eskandarian
UNC Chapel Hill
saba@cs.unc.edu

***Abstract*—We introduce new compression side-channel attacks against database storage engines that simultaneously support compression of database pages and encryption at rest. Given only limited, indirect access to an encrypted and compressed database table, our attacks extract arbitrary plaintext with high accuracy. We demonstrate accurate and performant attacks on the InnoDB storage engine variants found in MariaDB and MySQL as well as the WiredTiger storage engine for MongoDB.**

Our attacks overcome obstacles unique to the database setting that render previous techniques developed to attack TLS ineffective. Unlike the web setting, where the exact length of a compressed and encrypted message can be observed, we make use of only approximate ciphertext size information gleaned from file sizes on disk. We amplify this noisy signal and combine it with new attack heuristics tailored to the database setting to extract secret plaintext. Our attacks can detect whether a random string appears in a table with $> 90\%$ accuracy and extract 10-character random strings from encrypted tables with $> 95\%$ success.

I. Introduction

Compression side-channel attacks take advantage of the combination of compression and encryption to learn information about encrypted messages [16, 27]. Encryption methods used in practice aim to hide the content of a message, but do not aim to fully hide its length. Since compression post-encryption is ineffective, all systems that combine compression and encryption first compress the plaintext and then encrypt. This leaks information about encrypted plaintexts through the length of the resulting ciphertext [18, 22].

An attacker who can append their own input to data that is about to be compressed and encrypted can essentially insert "guesses" about the secret plaintext and determine whether those guesses are correct by observing the resulting ciphertext size. A correct guess is likely to compress well with the private data and result in a smaller ciphertext, while an incorrect guess is likely to compress poorly and result in a larger ciphertext. This intuition forms the backbone of the celebrated CRIME and BREACH attacks [16, 27], which enabled stealing cookies or CSRF tokens from encrypted HTTPS sessions. While the potential for compression side-channel attacks is understood to exist whenever systems combine compression and encryption, few works explore potential attacks beyond the TLS setting. Section VIII contains a brief summary of these works.

This paper introduces new compression side-channel attacks against databases. Modern databases provide rich support for data compression as well as encryption of data at rest. We demonstrate a number of settings where an attacker with limited, indirect access to an encrypted and compressed database table can extract arbitrary plaintext with high accuracy. Moreover, we show an even more rapid and extremely accurate attack that enables detection of specific adversary-chosen strings, e.g., email addresses, names, or keywords, in an encrypted table. We call our new attacks DBREACH attacks, standing for *Database Reconnaissance and Exfiltration via Adaptive Compression Heuristics*.

Our attacks require only that an adversary has the ability to insert and update rows in a victim table, either directly or through, e.g., a web interface, and that the attacker can learn the size of encrypted tables. We discuss various settings where an adversary has such capabilities when we elaborate our threat model in Section III.

We instantiate our attacks against MariaDB [2] and MongoDB [3], configured to use the default InnoDB [1] and WiredTiger [24] storage engines, respectively. We also demonstrate the feasibility of our attacks against the InnoDB variant in MySQL [5], using a slightly stronger threat model.

The storage engine is the component of the database system that handles the actual reading, writing, and updating of tables. Our attacks apply to all three compression schemes supported by MariaDB's standard InnoDB installation, two of which are also supported

by MongoDB.

Extending the generic notion of compression side channels to a concrete attack on databases requires solving new challenges not present in other settings. Since database pages are ultimately stored in a filesystem, access to file sizes provides only a low-resolution signal regarding the actual size of the underlying table. The database's choices of how to lay out table data on disk pose an additional challenge. These obstacles render any naïve use of the techniques of CRIME, BREACH, or related attacks ineffective.

We introduce new attack techniques that correctly align an attacker's "guess" with the page boundaries of both the database storage engine and the underlying filesystem, significantly boosting the noisy signal provided by file size information. We also design a series of new heuristic approaches to using guess compressibility for plaintext extraction. We implement attackers that retrieve file size information from the filesystem and insert guesses using either direct, unprivileged access to the DBMS or via a web front-end.

Our attacks can detect whether a random string appears in a table with at least 85% accuracy for any of the compression algorithms analyzed, even achieving accuracy of over 99% under certain conditions. Even for structured string data where the compression side channel is noisier due to incidental compression, e.g., English text or email addresses, we can detect the presence of a string with 70% or higher accuracy. Moreover, for the default zlib compression algorithm in MariaDB, we can extract 10-character random strings with greater than 95% success, and 17-character strings with 92% success.

Our proof of concept code is available at https://github.com/mathewdhogan/dbreach-code.

Disclosure. We discussed DBREACH and potential mitigations with the security teams of MariaDB, MongoDB, and MySQL. Since the attack is against the general encryption and compression protocol rather than an implementation bug, a patch was not deemed necessary or feasible.

II. Background

This section introduces the notion of a compression side channel and gives the background on compression and databases that we use to develop our attacks in subsequent sections.

A. The Issue with Compression and Encryption

Modern encryption schemes aim to guarantee *Semantic Security* [18]. Informally, semantic security means that one cannot feasibly extract any information about the original plaintext from an encrypted message, except for its length. Applying compression to the plaintext, prior to encrypting it, breaks the guarantee of semantic security, as the length of the message now depends on the compressibility of the plaintext. By observing the length of an encrypted and compressed message, an adversary learns additional information about the entropy of the plaintext, contrary to the definition given above.

This observation, first explored by Kelsey [22], resulted in critical web vulnerabilities in the form of the CRIME [27] and BREACH [16] attacks. These attacks suggest that anywhere compression and encryption are combined, we should consider whether this combination poses a practical security threat that can be exploited. In this paper, we will show how to abuse compression to extract plaintext from encrypted databases. The rest of this section provides the background on encryption and compression in the InnoDB and WiredTiger storage engines that will be needed to explain our attacks.

B. Storage Engines

We instantiate our attacks against two storage engines: InnoDB [1] and WiredTiger [24].

InnoDB. The InnoDB storage engine [1] is used by the relational database management software (RDBMS) MariaDB [2] and MySQL [4], although the two InnoDB versions have diverged in terms of implementation and functionality. We attacked InnoDB as used by both MariaDB and MySQL. That said, differences in MySQL's implementation led to a slightly less powerful attack than that against MariaDB. We discuss the MySQL attack in more detail in Appendix D; for the rest of this paper, we will use "InnoDB" to refer to MariaDB's version of the storage engine, unless otherwise noted.

WiredTiger. We also demonstrate our attacks against the WiredTiger storage engine [24] used by the RDBMS MongoDB [3]. We selected WiredTiger and MongoDB as an additional target because it differs from InnoDB in several important ways, including in its fundamental storage paradigm: WiredTiger and MongoDB are NoSQL database platforms, unlike InnoDB. The success of our attacks against such a variety of storage software and paradigms demonstrates their broad generalizability.

C. Table Storage

InnoDB for MariaDB offers two choices for storing tables: a single shared tablespace or separate file-per-table tablespaces. For simplicity, we will only consider the latter case, which is the default on a standard Ubuntu

installation of MariaDB. In this setting, the data for each table in a database resides in a separate file on disk.

Due to its distinct storage paradigm, WiredTiger uses different database terminology: tables are "collections," and the terms "row" and "column" are eschewed in favor of "document" and "field" respectively. For readability, we will use the more traditional database terms throughout this paper, even when referring to WiredTiger. Despite the differences in terminology, WiredTiger also stores the data for each table (i.e., collection) in a separate file on disk.

In both InnoDB and WiredTiger, each table file begins with some table metadata, followed by the table contents, with each row preceded by a row header. In WiredTiger, the value of each column is also preceded by a header containing the column name. We find that insertions in both storage engines are generally written to disk sequentially and that updating the contents of a row with new contents of similar size often overwrites the original row in place. There are exceptions to this general behavior, e.g., in cases of updates that cause a row to grow or insertions that overwrite space freed by previous deletions, but the behavior is consistent enough for our purposes. For brevity, we omit a longer discussion of the behavior of each storage engine. We do not discuss other aspects of how tables are stored on disk, e.g., indexes, as they will not be relevant to our attacks.

Table file growth. As the amount of data in a table increases, both InnoDB and WiredTiger lazily allocate space on-demand, e.g., during an insertion, if there is no space available in the existing table file for the new data. In both storage backends, the table file grows coarsely, with the engine allocating new file space in large chunks. When additional space is allocated, InnoDB adds a complete InnoDB database page at once, which is 16 KB by default. The default allocation size for WiredTiger is 4 KB. This means that, while using either storage engine, a table file's (uncompressed) size will stay constant through many insertions, and then increase suddenly by the default allocation size. Additionally, since deleted data is not actually removed from a table file, uncompressed table files never shrink unless a database administrator (DBA) runs the `OPTIMIZE TABLE` command (for MariaDB) or the `compact` command (for MongoDB). This implies that a table's size on disk provides only an imprecise approximation of the table's logical size in the database.

Encryption at rest. Both InnoDB and WiredTiger support the at-rest encryption of table files. Innodb utilizes AES in CBC or CTR mode; we use CTR as this is the recommended setting. This is implemented via a two-tier key hierarchy, where a 128-, 192-, or 256-bit root key encrypts a number of 128-bit table keys, each of which encrypts a table's 16 KB database pages before writing those pages to disk. WiredTiger uses 256-bit AES CBC or AES GCM encryption via OpenSSL [6]; we use the default CBC. WiredTiger leverages a 96- or 192-bit master key to encrypt its 256-bit database keys, which is then used for each table in the database. In both storage engines, encryption is the final operation before a page is written to the filesystem, and, importantly, it occurs after any compression has already taken place.

D. Compressing tables

Both InnoDB and WiredTiger provide several compression options that can be specified at table creation or as a database-wide config.

InnoDB. InnoDB provides several options for compressing tables, including the `COMPRESSED` row format and InnoDB page compression. We developed our attack against page compression, the recommended compression method according to MariaDB documentation.

In InnoDB Page Compression, each 16 KB page is compressed on its own before being written to disk. This means that any data on a given page will be compressed with all other data on that page, including deleted or updated rows that have not yet been fully overwritten.

WiredTiger. By default, WiredTiger enables block compression for all tables. The default compression window is equal to WiredTiger's allocation size, so typically 4 KB. Like InnoDB, all data within the 4 KB is compressed together, including not-yet-overwritten stale data.

Compression algorithms. Both InnoDB and WiredTiger support several different compression algorithms, with InnoDB supporting up to six and WiredTiger up to three. We demonstrate that our attack works against the following three compression algorithms that come pre-installed with MariaDB on Ubuntu, all of which are based on the LZ77 compression algorithm [35]:

- **zlib** [15]: A commonly used library that uses the `DEFLATE` storage format [11], and the most aggressive of the three compression algorithms. `DEFLATE` combines LZ77 compression [35] with Huffman encoding [19]. Huffman encoding introduces noise into the compression side channel (as previously observed by [27] and [16]) and makes *zlib* a more challenging algorithm to attack.
- **LZ4** [34]: An algorithm in the LZ77 [35] family. Less aggressive than zlib, LZ4 uses only LZ77 compression and no Huffman encoding.

- **Snappy** [8]: An open source compression library built by Google. Based on LZ77 [35], *Snappy* prioritizes speed over compression ratio and is the least aggressive of the three algorithms.

Notably, MySQL provides built-in support for zlib and LZ4, and MongoDB provides built-in support for zlib and Snappy. Thus, each of the three compression algorithms has at least two points of comparison across the storage engines that we attacked. The consistency with which we achieve high accuracy against a variety of compression algorithms and implementations suggests that the attack's success is not dependent on the precise details of how a storage engine's compression works.

Compression and file sizes. Although page compression reduces the size of an InnoDB page's contents, InnoDB writes the compressed page to disk with the original size of the uncompressed page. Instead of allowing for variable-sized database pages, InnoDB relies on the underlying filesystem to save space using *sparse files* [33] and *hole punching* [7]. WiredTiger similarly returns unused table file space to the filesystem, freeing 4 KB blocks once they are no longer needed to store data.

The freeing of compressed data has one very important consequence: although InnoDB and WiredTiger never explicitly remove uncompressed database pages, changes in a table file's compressibility can now lead to pages being released to the file system. As such, when using compression, table files can be made to shrink if the contents of a table become sufficiently compressible.

III. Threat Model

This section elaborates the threat model assumed by our attack techniques, and the settings where they can be applied. Broadly, our threat model follows those of prior works like CRIME and BREACH [16, 27], which take advantage of opportunities on the web to create compression oracles and carry out a compression side-channel attack [22]. Our work follows this same pattern in the database setting.

There are two fundamental abilities needed for exploiting a compression side channel that our work shares with CRIME and BREACH.

- The attacker's ability to insert data close to, and in the same compression window as, victim data.
- The attacker's ability to measure the size of the resulting compressed data.

In the context of databases, this means that a DBREACH attacker needs the ability to insert and update data near to victim data and the ability to measure the size of database tables.

To satisfy the first capability, we require that the attacker can insert and update database content, either directly as a regular (unprivileged) user using SQL queries, or via a frontend web interface or API that communicates with the database as its storage backend. We demonstrate attacks using both methods; details are included in Section VI. Note that it is possible for an unprivileged database user to have `SELECT` and `UPDATE` permissions on just some columns of a table, allowing her to insert and update rows in a table despite lacking permissions to read the contents of other table columns. DBREACH allows such users to read from the remaining table columns despite lacking the necessary permissions.

In Appendix D, we also show how to instantiate our attacks in a setting where the attacker has the additional ability to *roll back* file state; we use this altered threat model to attack MySQL's InnoDB.

The easiest way to assess the size of the compressed table, our second attack requirement, is to read the table file's on-disk size. This requires read access to the server machine on which the database is running. If the database uses table encryption at rest, an adversary who compromises the encrypted database server will be able to access a file's size but not its plaintext contents.

Note that our attacks rely a great deal on the proximity of victim data to attacker-controlled rows in the table file, as only data within the same compression window will compress together. This means that a DBREACH attacker must either target a database shortly after the insertion of desired victim data or spray the table with attacker-controlled data over time to enable later attacks that target any portion of the table.

IV. Plaintext Detection and Extraction Attacks

This section elaborates our different attack variants and explains the techniques behind them. We implement three varieties of plaintext guessing and extraction attacks:

1) a *Decision Attack*, where we determine whether a given string resides in the table,
2) a k*-of-*n *Inclusion Attack*, where we determine, out of a list of n strings, which k are most likely to be in the table, and
3) a *Character-by-Character Extraction Attack*, where we extract a whole string from the table without prior assumptions regarding table contents, other than a known prefix or suffix present in the table.

Each attack will involve the attacker having a set of "guesses" of what might be in the table and checking how likely each guess is to be correct. In practice, this

list could be a list of email addresses, names, or a list generated by adding a single character to a known prefix. To determine how likely a given string is to appear in a table, we introduce the notion of a *compressibility score*. Compressibility scores capture how compressible a database page becomes after the attacker inserts a guess, and a lower compressibility score indicates that the table is more compressible. The scores act as a proxy for the likelihood that the inserted guess string already appears in the table, since duplicated guesses will compress better than other guesses. We will begin by describing each of our attacks given access to compressibility scores for guessed strings before going into the details of how to compute scores in Section V.

Note that the techniques described here and in Section V apply to all compression algorithms which we evaluated; our attack is agnostic as to how precisely the table data is being compressed.

A. Plaintext Detection Attacks

We begin by describing our decision and k-of-n attacks, which detect if some plaintext appears in a table. These attacks build on the "String Presence Detection Attacks" that Kelsey theorized in [22]; we draw inspiration from Kelsey's high-level attack strategies to design and implement real, functioning attacks on InnoDB and WiredTiger.

Naïve attempts fail. A straightforward approach to the decision attack would involve picking a fixed threshold and determining that a string exists in a table if the compressibility score for that string falls below that threshold. Likewise, a simple k-of-n attack could rank all guessed strings by their compressibility scores and return the k strings with the lowest scores.

These naïve attempts fail to detect table contents. This is because compressibility scores give relative, not absolute, information on table compressibility, and other table contents may affect a guess's score, regardless of whether the guessed string appears in the table.

For example, one particularly challenging instance of this problem appears in what we call the "substring problem." This occurs when a string that occurs in the table is a substring of a guess, e.g., if the attacker is checking whether "Whitfield Diffie" occurs, and in reality only "Whitfield" does. In this case, we will observe low compressibility scores for the string, since the substring portion will compress entirely.

Our approach. The fact that that compressibility scores only provide relative information about string compressibility means that we cannot depend on a score in isolation to assess whether a string appears in a table. We solve this problem by acquiring two additional reference scores c_{yes} and c_{no} that inform us of what a compressibility score should look like for a string that is or is not in the table, respectively.

We can compute reference scores for inclusion or exclusion in the table by finding the compressibility scores of guess strings g_{yes} and g_{no} that we know appear or do not appear in the table. However, the goal of the decision attack is to determine whether a string exists in a table *without* prior knowledge of the table contents. Nonetheless, we can take advantage of the fact that a sufficiently long random string only appears in a table with negligible probability; thus, we select a uniformly random string for g_{no}.

While a random string suffices for g_{no}, determining g_{yes} still poses a problem. Our solution takes advantage of the nature of our compressibility scoring algorithm (see Section V). In the process of computing compressibility scores, we insert "filler" strings into the table to align guesses with database page boundaries. Because we have inserted the filler strings ourselves, we know that they appear in the table. Thus we can bootstrap our guess g_{yes} on top of the compressibility score calculation itself.

We require that g_{yes} has the same length as the real guess string(s), since a longer string match will result in greater compression and therefore lower compressibility scores. Thus, while running attacks on strings of varying lengths, we calculate a separate c_{yes} for each length.

With g_{yes} and g_{no} determined, an attacker can compute their compressibility scores to get c_{yes} and c_{no} as well as the compressibility scores $c_1, ..., c_n$ for the actual guesses $g_1, ..., g_n$ whose presence or absence in the table she wishes to discern. For each string, the attacker decides whether or not the string is present based on its score's proximity to c_{yes} relative to c_{no}. This is achieved by computing the decision value

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

which gives $d_g = 1$ if $c_g = c_{\mathsf{yes}}$ and $d_g = 0$ if $c_g = c_{\mathsf{no}}$.

For a k-of-n attack, we sort guesses by their decision values and pick the k highest values. This allows us to compare variable length strings while deciding if a guess is present in the table. For the decision attack, we don't require strings to precisely match c_{yes} due to the presence of noise in the side channel; rather, we consider strings with decision values above some empirically determined threshold t_{yes} to be in the table.

Choosing t_{yes}. Noise in the compression side channel can make two exact string matches have slightly different

compressibility scores. Thus, a successful decision attack requires properly setting the threshold parameter t_{yes}, which amounts to a tradeoff between minimizing the rates of false positives and false negatives.

The optimal value for the decision threshold t_{yes} may vary depending on the precise setting of the attack. As such, the best way to determine the ideal threshold value is to do so empirically based on the set of guesses an attacker wishes to check. Using this set of guesses, an attacker can repeatedly simulate the target database offline to generate a set of training examples consisting of compressibility scores and labels. An attacker can insert some random subset of their guesses into the offline testing table and then run the attack on all guesses, gaining data for what compressibility scores look like in the positive and negative cases. To generate a robust set of training examples, an attacker can vary the number of records on the page during data collection and can augment their set of guesses with additional similarly structured strings. Using the resulting training data, an attacker can simply search over many possible threshold values to find the optimal one.

Using this select methodology led to high levels of accuracy while executing the decision attack, as shown in Section VI-B. That this method is so successful suggests that the ideal threshold value depends mainly on the general structure of the target plaintext, rather than its precise contents. This threshold selection strategy does require the attacker to have some idea of the general structure of the data in the target plaintext; that said, this practical requirement was already necessary in order to effectively make use of any of our plaintext detection attacks.

B. Character by Character Plaintext Extraction

The character-by-character attack extracts arbitrary plaintext from an encrypted table one character at a time. This extractive attack is most similar to the CRIME and BREACH attacks instantiated against TLS [16, 27]; while details differ, our recursive, prefix-based approach borrows from the strategies of these prior works.

Extracting data one character at a time introduces new challenges not present in detection attacks. For example, choosing a random g_{no} no longer reliably works when the string is too short. Moreover, many characters will appear at different places in a table, meaning that determining whether or not a single character appears in a table does not really help to determine table contents.

Fortunately, the difficulties of character-by-character extraction can be turned into advantages. Since there are only a limited number of potential characters in a victim table, we know that one of these characters *always* appears at each position in the table. Thus, if only we had a way to fix the position in a table whose compressibility we are testing, we can do away with c_{yes} and c_{no}, and determine that the guessed character with the lowest compressibility score is the actual character in the table.

Absent a mechanism for fixing the position against which to test compressibility, many individual characters are likely to have the same compressibility score because they will appear at multiple places in each database page. Observe, however, that once a character by character attack is partially completed and several consecutive characters have been extracted from a table, this problem goes away. Instead of computing the compressibility score of one character at a time, the attacker computes the compressibility of the whole extracted string with one additional guessed character added to it.

In order to commence character extraction, we can bootstrap this process by taking advantage of additional attack scenario context. For example, suppose we wish to extract email addresses or US social security numbers (SSNs) from a table. Many SSNs begin with deterministically chosen digits, with most of the entropy stored in the latter half of the number. Likewise, many email addresses end in "@gmail.com" or another well-known domain. We can use such contextual knowledge to provide an initial substring to which we can add as the extraction attack progresses. The attack then proceeds as follows.

1) Begin with some known prefix (or suffix) p and alphabet of possible characters Σ.
2) Generate a candidate string list L of size $n = |\Sigma|$ by appending each element of Σ to p.
3) Compute a compressibility score for each element of L, and pick the one with the lowest compressibility score as the most likely candidate out of the n options. Call this most likely string p'. Note that $p' = p \| c$, where c is the most likely next character.
4) Set $p = p'$, and return to step 2.

To determine when to end the extraction process described above, an attacker can either rely on *a priori* knowledge regarding the plaintext length, or can terminate the process when no clear winner emerges from step 3, i.e., every option receives a very similar compressibility score.

V. Computing Compressibility Scores

We now describe our algorithm for computing compressibility scores. These scores enable the attack algorithms described in Section IV by providing information about how different guesses compress with extant table contents.

A. Finding a Page Boundary

To assess the likelihood that a particular string is present in the target table, we need a way of quantifying how compressible the table page is when the attacker has inserted that string as a guess. Recall, however, that the extent of table compression in InnoDB and WiredTiger does not directly reflect the file size on disk (see Section II). Instead, table sizes increase in discrete chunks, only when all available space has been exhausted. Similarly, table sizes decrease only when enough space has been cleared to free an entire database page. This means that compressibility scores cannot be computed by simply observing file sizes, as was done with network messages in previous compression side-channel attacks [16, 27]. Instead, we need a way to learn how compressible strings are despite the limited granularity of table size measurements. This means we must use our ability to manipulate a table to gain as much information as possible from discrete jumps in table size.

As a first step, we need to make sure that our compressibility scoring efforts start from a common baseline where inserting a more compressible string will cause a table to be observably smaller in size than inserting a less compressible string. To do this, we insert "filler rows" into the table until the compressed table is right on the brink of growing in size. In each of these filler rows, `VARCHAR` and other text fields should be filled with a character set that is (wherever possible) disjoint from the set of characters actually in the table; this is to avoid introducing noise into the side channel due to compression between filler rows and other table contents. Appendix B describes the structure of filler rows in the implementation of our attacks.

While inserting filler rows, we have a choice: we can either insert sufficient filler data such that the table grows in size (and then immediately stop inserting rows), or we can insert filler content such that the table is *about* to grow in size, by causing the table to grow and then shrinking it again by reducing the amount of incompressible filler. In our implementation, we take the former approach with InnoDB and the latter approach with WiredTiger. In either case, the end of the table is now very close to the "page boundary" at which point the table size changes; our compressibility scores approximate the signed distance from the end of the table to this page boundary.

B. Assessing Compressibility

Having a table at the brink of changing size allows us to compare many guesses against each other by repeatedly inserting a guess and estimating the distance from the end of the table file to the point at which the table changes size. The process of filling a table to the boundary between two page sizes only needs to run once and can be used for many compressibility score evaluations.

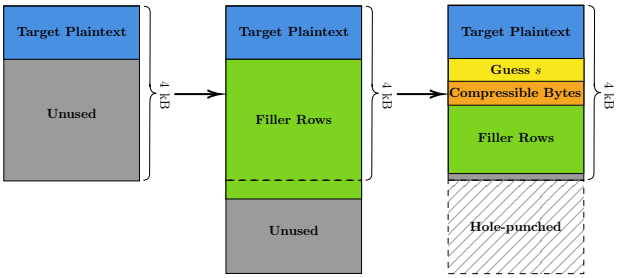

Fig. 1: Table state throughout the compressibility scoring algorithm. The table begins with target plaintext on the most recently allocated, and partly unused, page. First, the attacker inserts filler rows to overflow onto the next page. Next, the attacker inserts the string s into the first filler row and adds compressible bytes one at a time until the new page is hole-punched, shrinking the table back to its earlier size.

Compressibility scores are integers, with smaller scores (including more negative scores) indicating more compressible strings. After the initial "filler row" setup stage, our approach to computing compressibility of a string s proceeds as follows.

1) We update the first "filler row" to contain the string s without changing the total row size. That is, we overwrite the first $|s|$ bytes of the filler row with s. Note that if there are multiple text columns per row, s does not need to be in the "correct" column, i.e., the actual column where it occurs in the table, because all columns are compressed together.
2) We now seek to measure the distance from the current end of the table file to the point at which the file changes in size. If we added filler bytes until the table grew (InnoDB), then we start from the second filler row and, going byte by byte, overwrite the incompressible filler data with compressible bytes. In practice, this just means replacing each random filler byte to be the same character.
   If we added filler bytes until the table was *about to grow* (WiredTiger), we start from the second filler row and, going byte by byte, overwrite the compressible bytes with incompressible filler data. In practice, this just means replacing each identical compressible character with random filler data.
3) We stop when the on-disk table size changes due to shifts in compressibility. Let the compressibility score c_s be the number of compressible bytes that

were added to change the table size. If compressible bytes were replaced with incompressible bytes in order to grow the table, then c_s will be negative.

The compressibility score c_s now serves as a proxy measurement of the signed distance from the end of the table (after inserting s) to the point at which the table changes in size. This measurement allows us to assess the overall compressibility of the table with each guess inserted.

The intuition behind this compressibility algorithm is that if the string s is present in the table, then some compression will take place after s is inserted in step 1. Thus, the table will be smaller with s inserted as compared to some non-present string, so strings that occur in the table will have lower compressibility scores.

After each compressibility score computation, we update the table so that all filler rows once again contain the original incompressible filler data, effectively resetting the table state so it can be used to compute another compressibility score. A visual representation of this compressibility scoring algorithm can be seen in Figure 1.

C. Obstacles to Measuring Compressibility Precisely

In practice, noise in the compression side channel can lead to compressibility scores that fail to precisely capture the true compressibility of different guesses with the target plaintext. Sources of noise include the following.

- Unexpected compression due to shifts in the Huffman encoding of the table,
- Unexpected compression with other parts of the table file,
- Compression within the string itself, e.g. the string "abcabcabc" would appear more compressible than a string like "abcdefghi," even if neither is in the table, and
- Unexpected insertions onto different pages due to a fragmented table representation on disk, e.g., in a table that has had many deletions.

Noise in the side channel will occasionally lead to inaccurate compressibility scores and false positive results in our attacks. In practice, these false positives are relatively rare for large enough strings, in both storage engines. That said, in Section VI, we observe that the compression signal is less noisy and more reliable while attacking WiredTiger as compared to InnoDB. This is likely because WiredTiger's simpler NoSQL storage format involves storing less mutable metadata that can interfere with our compression.

If the strings being evaluated are short, or differ by only a few characters, noise can make our attack noticeably less accurate. This concern is especially prescient for the zlib compression algorithm, which is the only one that uses Huffman encoding [19].

Some of these issues, especially those dealing with Huffman coding, have been previously mentioned by the authors of BREACH [16], who propose mechanisms for de-noising compression side channels. We find that our compressibility scoring scheme, when augmented with the optimizations in Section V-D, gives good results despite the remaining potential for noise in the side channel.

D. Amplifying Compressibility with Repetition

Although a naïvely computed compressibility score can give inaccurate results due to a noisy side channel, truly more compressible strings do score better on average. As such, we can amplify the accuracy of our compressibility scoring through repetitions of the algorithm. In Section VI, we report on the relationship between the number of repetitions used and accuracy of character by character extraction. Surprisingly, repetition suffices for high-accuracy compressibility measurement even without using other noise-mitigation optimizations present in prior work.

After measuring the number of bytes b_{s_i} we need to change to compress the table for all guesses $s_i, i \in \{1, ..., n\}$, we repeat the compressibility scoring from scratch, starting with new random incompressible filler data, and measure compressibility again. After completing r repetitions we have counts $b_{s_i}^{(1)}, b_{s_i}^{(2)}, ..., b_{s_i}^{(r)}$ for $i \in \{1, ..., n\}$. The way we use these values to compute amplified compressibility scores varies somewhat based on the underlying compression algorithm.

Amplified compressibility for zlib. One way to compute an amplified compressibility score would be to sum scores for each s_i, i.e., to compute

$$c_{s_i} = \Sigma_{j=1}^{r} b_{s_i}^{(j)}.$$

Recall, however, that compressibility scores only give information about the *relative* compressibility of strings, not absolute information. Thus the compressibility of a given string may vary widely between repetitions, and compressibility scores in different repetitions of the scoring algorithm cannot be compared directly. We avoid this problem by normalizing scores in each round. We do this by calculating the difference between the number of bytes needed to compress a guess and the minimum number of bytes needed to compress a guess in that repetition. That is, we set

$$c_{s_i} = \Sigma_{j=1}^{r} \big(b_{s_i}^{(j)} - \mathsf{Min}_{\ell=1}^{n} (b_{s_\ell}^{(j)}) \big).$$

Target Data	Random	English	Emails
Snappy (IDB)	.50 (.93)	.62 (.74)	.85 (.80)
zlib (IDB)	.45 (.92)	.69 (.79)	.81 (.90)
LZ4 (IDB)	.47 (.99)	.55 (.83)	.78 (.91)
Snappy (WT)	.35 (1.00)	.39 (.90)	.49 (.91)
zlib (WT)	.46 (.93)	.35 (.83)	.52 (.92)

TABLE I: Best accuracy threshold (and average accuracy achieved at that threshold) for each compression algorithm, storage engine, and target data setting. IDB denotes InnoDB, whereas WT denotes WiredTiger.

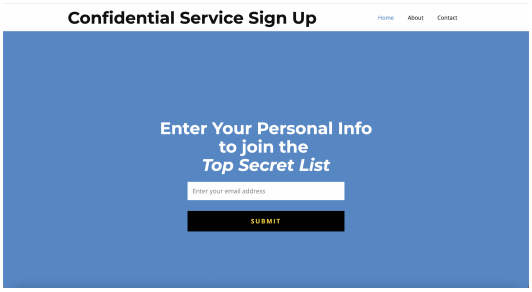

Fig. 2: Frontend for example web interface through which we instantiated our k-of-n attack. Web interface was backed by a MariaDB/InnoDB database instance in which users' sensitive data was compressed and encrypted.

This method for calculating c_{s_i} proves effective for amplifying the signal in zlib compression, allowing us to overcome the noise added by Huffman encoding. We detail additional heuristic techniques for calculating more precise compressibility scores for LZ4 and Snappy in Appendix C.

We find that repetition is indispensable for our character by character attack, whereas our other attacks do fairly well even without repetition. This is because selecting one incorrect character renders the prefix incorrect and dooms the rest of the attack. As such, we expect to witness an exponential decay in the likelihood of fully extracting a string as the target string's length increases; if we have a probability $0 \leq q \leq 1$ of correctly guessing the next character of a string, then we expect to have probability q^n of extracting an entire string of length n. We empirically witness this decay relationship between the length of the target string and the likelihood of success in Section VI-C.

VI. Evaluation

Attack environment. 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 10.3.29 and MongoDB Enterprise Server 6.0.0 on top of Ubuntu 20.10.

To demonstrate that our attack can be instantiated remotely through a web interface, as discussed in Section III, we attacked our own database server using an InnoDB-based REST API that transmits form data from the web frontend seen in Figure 2. Further details on this proof-of-concept attack are included in Appendix E.

We also instantiated our attacks using Python connectors for MongoDB and MariaDB, running locally on the database server and using only the unprivileged database access allowed by our threat model. To simplify and quicken evaluation, the experiments described here were conducted locally in this manner, thus avoiding network overhead.

In each of these experiments, we targeted the final, partially empty database page and configured our database to flush changes to disk after every insert or update. A database with lower flushing frequency would slow down our attacks by a factor proportional to the flushing delay but would not affect the accuracy of our attacks. In this case, an attacker would simply have to wait between operations for the changes to be reflected on disk.

Evaluation datasets. In most of our experiments, we assess the accuracy of the attack variants on three different types of target plaintext: randomly generated text, English words, and email addresses. Each dataset consists of a simple `(id, value)` schema, where `id` is an integer and `value` is a string. The random plaintext was generated by selecting random lowercase characters to form 2,000 strings between 10 and 20 characters in length. English words were sampled from a list of the top 10,000 English words [12], discarding words less than 9 characters in length (resulting in 2,241 words). Lastly, 2,000 synthetic email addresses were randomly generated using a random email address generator [26] which combines alphanumeric names with an assortment of email provider domains. Since many of our email addresses come from the same provider, repeated substrings across different entries are most common in the email dataset; on the other hand, repeated substrings of substantial length are very rare in the random dataset.

A. Decision Attack

We have previously mentioned that the Decision Attack relies on a threshold value t_{yes} while deciding whether a given guess is in the table. To assess the accuracy of the decision attack, we first use the procedure described in Section IV-A to determine the optimal

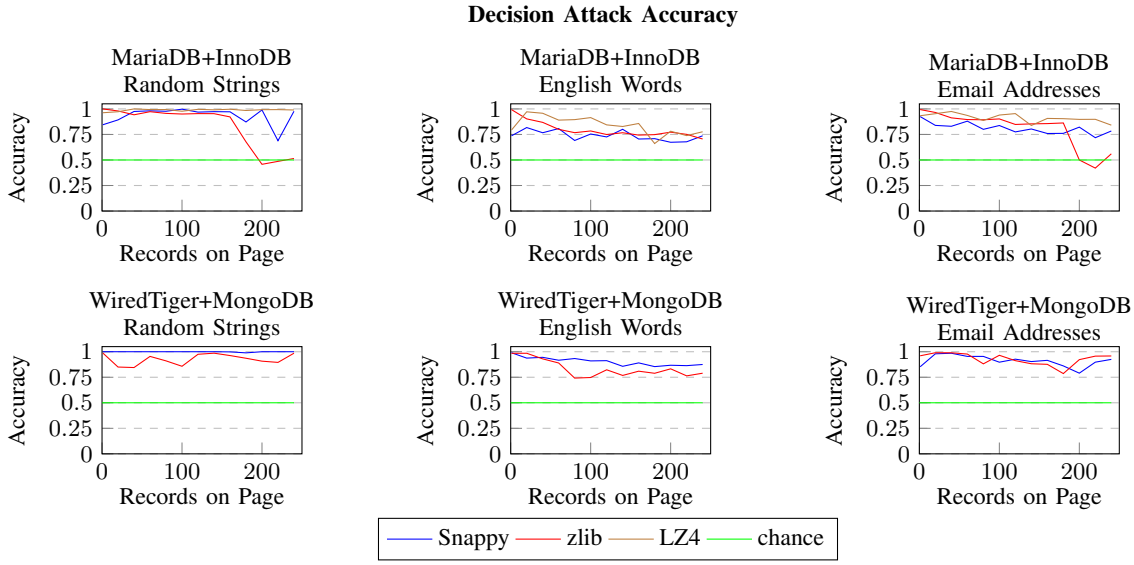

Fig. 3: Accuracy of the decision attack on each dataset. Our attack results in a high probability of determining whether a string is in the table across all three datasets, with a higher accuracy when the strings are random and the compression process is therefore less noisy. The "chance" line indicates the expected results of randomly guessing, which achieves an accuracy of 0.5. Sudden accuracy drops for large numbers of records are due to guesses spilling over onto a second page.

threshold t_{yes} for our attacks. The resulting best threshold for each evaluation set and storage backend appears in Table I. As expected, we found that the optimal threshold value varied depending on whether we were extracting random text, English words, or email addresses. Specifically, extracting text in which repeated substrings occur more frequently generally necessitates a higher threshold value to avoid false positives. Observe that, for each compression algorithm, the threshold values for each dataset are roughly ordered based on the likelihood of seeing repeated substrings in that text, with the random dataset obtaining the lowest thresholds and the email address dataset obtaining the highest thresholds. We also note that InnoDB typically has higher threshold values than WiredTiger for the same compression algorithms. This is due to a higher level of overall noise in the InnoDB compression side channel, leading to incorrect guesses occasionally having higher scores. More details on threshold selection are presented in Appendix A.

Next, we assess the accuracy and efficiency of the decision attack using the calculated threshold value. For these tests, we evaluate the decision attack on a test set of equal numbers of strings that do and do not appear in a table, so we would expect random guessing to achieve an accuracy of 0.5.

As seen in Figure 3, we observe consistently high extraction accuracies on each type of target text and for each compression algorithm, even as the number of records on the database page increases. While extracting random text, the easiest of the three evaluation datasets due to the lack of overlap between guesses, we achieve accuracies of over 90% for all three InnoDB compression algorithms up until there are 180 records on the database page. In the less-noisy WiredTiger environment we achieve even higher accuracy, including near-perfect 99.8% precision on the Snappy algorithm. This pattern of retaining high accuracy even on nearly-full pages, which we see on the other datasets as well, suggests that the decision attack algorithm is resilient even in noisy settings. The sudden drop-off in accuracy that occurs for the zlib algorithm on random strings and email addresses is due to the page becoming full; this is likely caused by the incompressibility of random text and the longer length of records in the email address dataset. Once the page fills, the guess and/or filler rows are inserted onto a different database page than the target plaintext in the table. Since compression only occurs within a database page and not between different pages, observing the compression signal in such a setting is not possible and is out of scope for our attack.

Regardless of the compression algorithm or the type of text being extracted, we found that the timing of the decision attack remained consistent. That said, the timing noticeably differed on WiredTiger and InnoDB. While

attacking InnoDB, setting up the attack, i.e., inserting filler rows until the table grows, took an average of 0.59 seconds with a standard deviation of 0.59 seconds. Running the attack took and an additional 1.6 seconds per guess (with standard deviation 1.3 seconds), including the time needed to compute reference compressibility scores. We observed slower times for WiredTiger: guesses took an average of 5.03 seconds, with standard deviation of 3.21 seconds. This slower guess time is likely due to the fact that WiredTiger is less well-suited for such a disk-flush-intensive workflow. Setup time took noticeably longer for WiredTiger, lasting an average of 184.71 seconds with a standard deviation of 125.93 seconds. The lack of substantial metadata chunks associated with subsequent row insertions made it take longer to find the page boundary in WiredTiger, leading to the longer setup time. The positive tradeoff is that the relative lack of per-row metadata allowed us to be more precise in finding the page boundary, potentially explaining some of WiredTiger's higher accuracy

The majority of the time spent on each guess is spent searching for the cutoff point of compressible bytes at which point the table changes in size. Since InnoDB or WiredTiger must re-compress and encrypt the entire page and flush to disk after each update, we are limited by the speed of the compression and encryption algorithms and, most consequentially, disk write speeds.

B. k-of-n Attack

To assess our k-of-n attack, we set k equal to the total number of records on the page, and observed the precision of the attack – the fraction of the k chosen strings that actually appear in the table. We set $n = 500$ and chose the 500 guesses at random from the strings in each dataset, of which k would be inserted into the victim table. The results from this experiment on each dataset and for both storage backends are shown in Figure 4.

We observed extremely high precision on random string extraction on both InnoDB and WiredTiger, with each compression algorithm achieving values over 90% for most values of k while attacking InnoDB and nearly perfect accuracy while attacking WiredTiger. Across the board, we achieve slightly higher precision while attacking WiredTiger due to its less noisy side channel, as discussed in Section V-C. The precision of the attack regressed slightly on the English word and email datasets due to the increased likelihood of compression between substrings of incorrect guesses and substrings in the target plaintext. Nevertheless, the attack outperforms chance in every case on both WiredTiger and InnoDB. The decrease in precision on the two structural datasets suggests that partial substring compression is the most consequential source of noise for this attack, whereas the similar precision achieved while attacking zlib and other (non-Huffman encoded) compression algorithms suggests that the noise from Huffman encoding is not consequential in this context.

As expected, running the k-of-n attack with larger values of n and the same value of k slightly increases the likelihood of false positives, since we are testing more incorrect guesses. This phenomenon can be seen in Figure 5, which shows the effect of varying n on attack precision while extracting email addresses from an InnoDB table that was compressed using zlib; we observed similar results for each compression algorithm and storage backend.

C. Character-by-Character Extraction

Since each guess in the character-by-character attack only differs from other guesses by a single byte, the sources of noise mentioned in Section V-C can have an out-sized effect on the outcome of the attack. As such, the amplification techniques described in Section V-D become very important in ensuring the accuracy of our character extractor. We now present evaluation results for the extraction attack against InnoDB using zlib, its default compression scheme. We present evaluation results for LZ4 and Snappy in Appendix C, after describing the compressibility score amplification techniques used for those compression algorithms. Overall, our character by character results for zlib are stronger than for LZ4 or Snappy.

To determine the optimal number of amplification rounds to run, we first measured the accuracy of the extractor at picking the correct next character given different amounts of repetition and varying length known prefixes. Each prefix was randomly chosen from the set of lowercase characters, as was the unknown next character which we attempted to guess. The results of this experiment, showing the accuracy of next-character prediction over more than 40 trials, are shown in Figure 6.

We observe prefix length has little effect on the accuracy of our character extractor, as long as the prefix is at least three bytes. Repeated strings of less than four bytes in length are too inconsistently compressed, preventing us from achieving high accuracies; when the known prefix is one or two bytes, we are only able to achieve a maximum accuracy of around 80% even with 40 amplifications. With a sufficiently large prefix of 3 bytes or above, however, we are able to achieve accuracies between 95% and 100% after 30 amplification rounds.

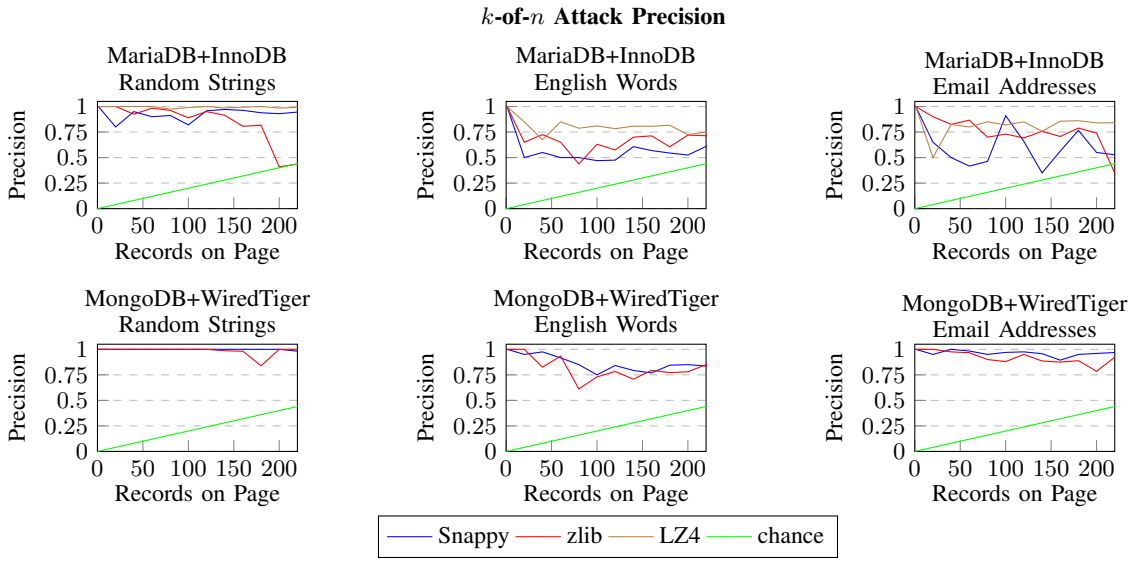

Fig. 4: Precision (fraction of correct guesses) of the k-of-n attack with $n = 500$ and k set to the number of records on the page. The "chance" line indicates the expected results of randomly guessing k of the n strings. Sudden precision drops for large numbers of records are due to guesses spilling over onto a second page.

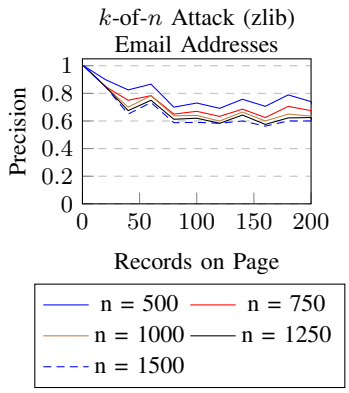

Fig. 5: Effect of increasing n on the precision of the k-of-n attack, where k is the number of records on the page.

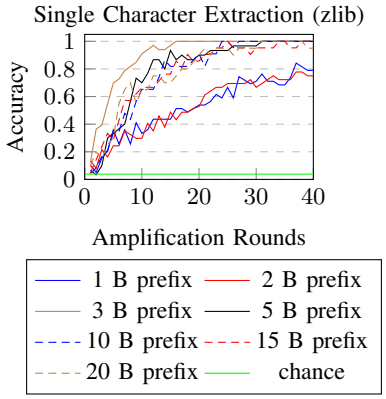

Fig. 6: Single character extraction accuracy. Amplification leads to high accuracy for prefixes of three bytes or more.

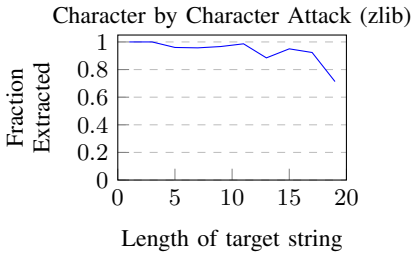

Fig. 7: Average fraction of bytes extracted by the character by character attack prior to making the first mistake, for different length target strings. Target string length does not include the 14-byte known prefix. Averaged over 20 trials per string length.

These strong next-character extraction results lead to high accuracy in the character by character attack. To assess the accuracy of the attack, we used it to extract different length strings given a known (randomly chosen) 14-character prefix and using 30 amplification rounds. The results from this experiment can be seen in Figure 7. We witnessed near perfect accuracy for short strings of length 10 or below, 92% recovery for strings of length 17, and an average of 72% recovery if the string is of length 20. This quick falloff in accuracy as the target string length increases confirms our intuition regarding the exponential increase in difficulty as plaintext length increases.

The timing of the character by character attack is affected greatly by the number of amplification rounds we must perform. While using InnoDB's zlib, we observed an average time per amplification round of 20.3 seconds, with a standard deviation of 10.7 seconds. Since a single round takes about 20 seconds, running 30 amplifications, as we did for zlib in Figure 7, takes approximately 600 seconds to extract a single character. In practice, an attacker could balance their need for accuracy with their need for speed by increasing or decreasing the number of amplification rounds.

We also instantiated a proof-of-concept character extractor attack against WiredTiger, but we focused our evaluation efforts on InnoDB, as InnoDB appeared to 1) be more noisy and therefore more challenging to attack in our earlier experiments, and 2) take less time for an amplification round than WiredTiger, enabling more extensive experiments.

VII. Mitigation

The first and most effective means of defense against DBREACH is to turn off compression entirely. This eliminates the compression side channel from which the attack arises, but may not be a feasible solution for storage space and cost reasons.

Short of turning off compression entirely, the threat surface can also be significantly reduced by not compressing together rows that may contain data from multiple users. A first step in this direction, usable today with no changes to existing database implementations, is to use MariaDB's storage-engine independent column compression, which compresses the contents of selected columns separately for each row. This reduces the chance that data from different users is compressed together, but it also dramatically reduces the effectiveness of compression.

Database implementations could consider adding support for full row-by-row compression to maximize compression within one user's entry in a table but disallow compression between different users' data. A more significant change would be to allow users to categorize data into separate compression buckets where only data in the same compression bucket can be compressed together. This would give users more granular control of what data can be compressed together, but it would involve larger changes in storage engine design and significantly complicate the user experience.

VIII. Related Work

The earliest work on compression side channels is by Kelsey [22], who introduced several methods for detecting the presence of strings and extracting plaintext information when compression is paired with encryption. Practical compression side-channel attacks against TLS and HTTPS were introduced in CRIME [27] and BREACH [16], and a number of subsequent improvements have either improved the precision or reduced the burden of launching such an attack [10, 17, 21, 23, 31]. Earlier works had taken advantage of size or timing information to compromise encrypted communications over a network, but they did not directly use a compression side channel [29, 32].

Concurrent with our work, Schwarzl et al. [28] have introduced timing side-channel attacks on memory compression, and among the example applications of their attack, they demonstrate an attack against one mode of compression in Postgres databases. However, their threat model differs from ours in that their attacks take advantage of decompression timing and require a table's contents to be read in order for the attack to work. Moreover, their attack on Postgres operates in a stronger attacker model where the attacker can modify one part of a single cell in a table but does not have read access to other parts of the same cell. While the attacks of Schwarzl et al. have implications in many other areas not covered by DBREACH (e.g., PHP applications using Memcached, Linux memory compression), their applications to databases are more limited in scope. Somewhat related, memory deduplication can also expose data via timing side channels that are observable to local and remote attackers. Compressed caches can also expose secret data, as shown by Tsai et al. [30] who develop techniques specific to attacking caches for the VSC architecture.

Several prior works have introduced defenses against compression side-channel attacks, particularly in the web context [9, 20, 25]. Broadly speaking, these defenses attempt to achieve the best possible compression performance without compromising security. CTX [20] proposes a "context-hiding" approach to ensure that sensitive data cannot be compressed with other data. Debreach [25] (not to be confused with our attack, DBREACH) uses static analysis techniques to detect and mitigate potential compression side channels.

Our primary contribution to the body of work exploring compression side channels is the identification and development of practical, performant, and accurate attacks against multiple commonly used database systems. As with other examples of exploiting compression oracles, this required carefully developing generalizable techniques that are specific to the databases setting where a naïve application of the general underlying principle would not work.

While no prior work has shown compression side-channel attacks against InnoDB, WiredTiger, or other database storage engines, others have shown ways to exploit other aspects of the InnoDB design, such as to extract plaintext or for forensic purposes, e.g., [13, 14].

IX. Conclusions

We have introduced DBREACH attacks – new compression side-channel attacks capable of extracting the contents of encrypted database tables. DBREACH attacks

detect and extract plaintext in tables that use compression and encryption, and they arise in various situations where an adversary posesses limited access to write to and update a table and observe table metadata. We demonstrated DBREACH attacks on the InnoDB and WiredTiger storage engines using a range of compression algorithms and storing several types of plaintext. Lastly, we discussed potential mitigations to prevent databases from falling victim to DBREACH attacks.

ACKNOWLEDGMENT

We would like to thank Dan Boneh for many helpful conversations. We are also grateful to the anonymous reviewers for their thoughtful comments on earlier versions of this paper.

This work was funded by NSF, DARPA, a grant from ONR, the Simons Foundation, NTT Research, and a Google Research Scholar award. Google additionally provided GCP credits used to perform our evaluation. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

REFERENCES

[1] InnoDB. https://en.wikipedia.org/wiki/InnoDB.
[2] MariaDB. https://mariadb.com/.
[3] MongoDB. https://www.mongodb.com/.
[4] MySQL. https://www.mysql.com/.
[5] MySQL InnoDB. https://dev.mysql.com/doc/refman/5.7/en/innodb-introduction.html.
[6] OpenSSL. https://www.openssl.org/.
[7] Punching holes in files. https://lwn.net/Articles/415889/.
[8] snappy. https://google.github.io/snappy/.
[9] J. Alawatugoda, D. Stebila, and C. Boyd. Protecting encrypted cookies from compression side- channel attacks. In *Financial Cryptography*, 2015.
[10] T. Be'ery and A. Shulman. A perfect crime? only time will tell. *Black Hat Europe*, 2013.
[11] P. Deutsch. DEFLATE compressed data format specification version 1.3. https://www.rfc-editor.org/rfc/rfc1951.
[12] first20hours. first20hours/google-10000-english. https://github.com/first20hours/google-10000-english.
[13] P. Frühwirt, P. Kieseberg, S. Schrittwieser, M. Huber, and E. R. Weippl. InnoDB database forensics: Reconstructing data manipulation queries from redo logs. In *Seventh International Conference on Availability, Reliability and Security, Prague, ARES 2012, Czech Republic, August 20-24, 2012*, pages 625–633. IEEE Computer Society, 2012.
[14] A. Futoransky, D. Saura, and A. Waissbein. The ND2DB attack: Database content extraction using timing attacks on the indexing algorithms. In D. Boneh, T. Garfinkel, and D. Song, editors, *USENIX Workshop on Offensive Technologies, WOOT*. USENIX Association, 2007.
[15] J.-J. Gailly and M. Adler. zlib. https://zlib.net/, 1995.
[16] Y. Gluck, N. Harris, and A. Prado. BREACH: Reviving the CRIME attack. *Black Hat*, 2013.
[17] T. V. Goethem, M. Vanhoef, F. Piessens, and W. Joosen. Request and conquer: Exposing cross-origin resource size. In *USENIX Security*, pages 447–462, Austin, TX, Aug. 2016. USENIX Association.
[18] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.
[19] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
[20] D. Karakostas, A. Kiayias, E. Sarafianou, and D. Zindros. CTX: Eliminating BREACH with context hiding. *Black Hat Europe*, 2016.
[21] D. Karakostas and D. Zindros. New developments on BREACH. *Real World Crypto*, 2016.
[22] J. Kelsey. Compression and information leakage of plaintext. In J. Daemen and V. 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.
[23] D. Krakostas, D. Zindros, E. Sarafianou, and D. Grigoriou. Rupture. https://github.com/decrypto-org/rupture.
[24] MongoDB. The WiredTiger Storage Engine. https://www.mongodb.com/docs/manual/core/wiredtiger/.
[25] B. Paulsen, C. Sung, P. A. H. Peterson, and C. 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.
[26] Randommer.io. Random email generator. https://randommer.io/random-email-address.
[27] J. Rizzo and T. Duong. CRIME. *Ekoparty*, 2012.
[28] M. Schwarzl, P. Borrello, G. Saileshwar, H. Müller, M. Schwarz, and D. Gruss. Practical timing side channel attacks on memory compression. *CoRR*, abs/2111.08404, 2021.
[29] D. X. Song, D. A. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In D. S. Wallach, editor, *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*. USENIX, 2001.
[30] P.-A. Tsai, A. Sanchez, C. W. Fletcher, and D. 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.
[31] M. Vanhoef and T. V. Goethem. HEIST: HTTP encrypted information can be stolen through TCP-windows. *Black Hat*, 2016.
[32] A. M. White, A. R. Matthews, K. Z. Snow, and F. Monrose. Phonotactic reconstruction of encrypted VoIP conversations: Hookt on fon-iks. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 3–18. IEEE Computer Society, 2011.
[33] Wikipedia. Sparse file. https://en.wikipedia.org/wiki/Sparse_file.
[34] Yann Collet. LZ4. https://lz4.github.io/lz4/.
[35] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

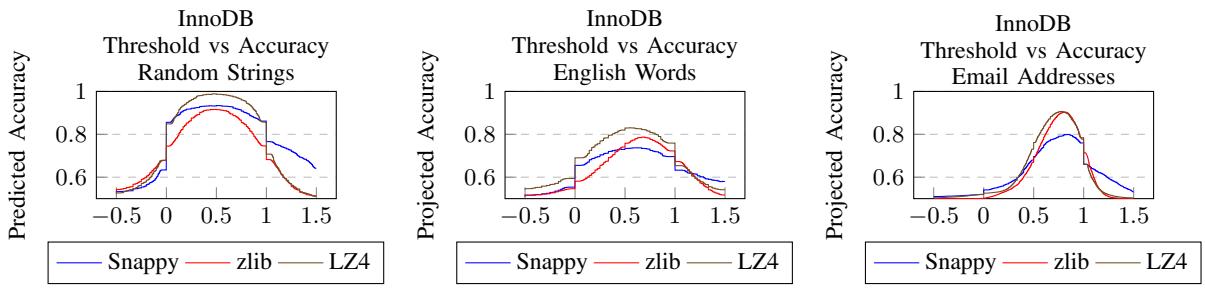

Fig. 8: Threshold selection data for each evaluation dataset while attacking InnoDB. Displays accuracy achieved on each training dataset under different threshold values. Note that the curves skew towards 1 for datasets in which repeated substrings are more common, since higher threshold values are needed to avoid false positives.

Appendix A
Finding the Optimal t_{yes}

We chose our optimal thresholds for the decision attack by trying every increment of 0.01 between -0.5 and 1.5 to generate the graphs of accuracy vs. threshold values for InnoDB seen in Figure 8. We generated the training dataset for Figure 8 using the methodology described in Section IV-A, and we excluded from our calculation any training data points in which the table page was too full to pull off the attack. (It is easy to detect such instances during data collection by monitoring the table file's size.) The maximum point in each graph is the threshold we chose to use in our evaluation; these maximal values can be seen in Table I. Trying decision thresholds below 0 and above 1 accounts for the possibility that a string may be more compressible than a string we know to be in the table or less compressible than a string we know to be absent from the table. As expected, there is a steep drop in projected accuracy of the decision attack if the threshold is below 0 or above 1.

We utilized the same process to calculate the optimal threshold values for WiredTiger as well, which generated similar-shaped graphs to those in Figure 8. For brevity, we only include the InnoDB graphs in the appendix.

Appendix B
Filler Row Format

This section specifies the structure of filler rows and the compressible bytes used to replace incompressible filler contents. A naïve implementation of would have random strings play the role of incompressible strings and then replace the contents of the random strings one character at a time with the same character, ideally one which does not naturally appear in the table to reduce the chance that the compressible bytes do not compress with other text already in the table. Our implementation uses "∗".

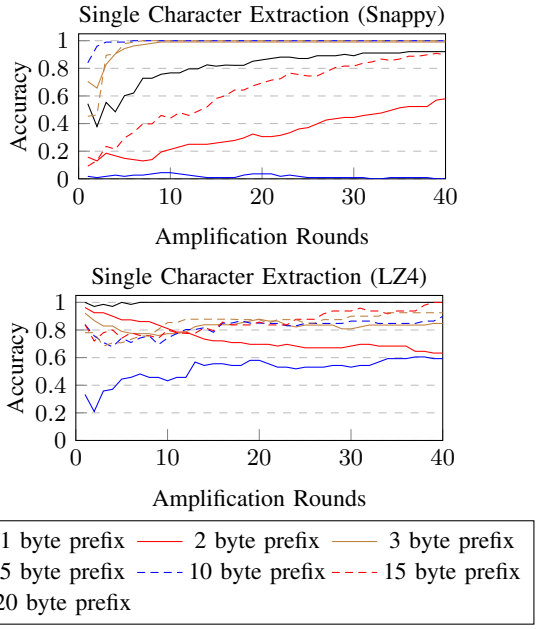

Fig. 9: Single character extraction results for Snappy and LZ4.

This approach works but is suboptimal because it results in a noisy measurement of compressibility. Real compression functions do not always compress a small string formed by repeating the same character many times. Luckily, a longer string consisting of all the same character will be much more likely to absorb an extra identical character without growing its compressed size. The details vary between compression algorithms, but we observed this same high-level behavior across all the compression algorithms tested.

Following these observations, we design our filler rows such that the replacement of one random character with a compressible character almost always results in the compressible character properly compressing with

the string that precedes it. After the first filler row, which has all incompressible contents (to be replaced by a guess string), our remaining filler rows contain a "bootstrapper" compressor string, which takes the form of many simultaneously repeated characters. For MariaDB, we use the "bootstrapper" string $*^{100}$, followed by incompressible random data to fill the rest of the row. This way, we can replace an incompressible byte with an additional asterisk and the string will very likely shrink in size as the compressed bootstrapper string absorbs another identical character. For WiredTiger, since row size is unlimited, we set the bootstrapper string to be even larger ($*^{5000}$ initially), and gradually shrink its size until the entire page is filled.

APPENDIX C
CHARACTER BY CHARACTER ATTACK FOR SNAPPY AND LZ4

In this appendix, we provide further strategies and evaluation for using the character-by-character attack against InnoDB using Snappy and LZ4. Primary details related to how the attack works and evaluation against zlib are found in the main body of the paper, in Sections VI-C and IV-B.

When using Snappy or LZ4 compression in a character by character attack, we modify our compressibility measuring scheme based on empirical observations of compression behavior. As expected, we find that the correct next character guess s^* in a character by character attack always has values $b_{s^*}^{(j)}$ that differ significantly from incorrect guesses, whose b-values tend to cluster together. However, due to idiosyncrasies of the underlying compression algorithms, the direction of the difference in scoring sometimes means that the value $b_{s^*}^{(j)}$ is sometimes considerably *higher* than other scores, rather than lower, as we would expect.

We correct for this behavior by switching the calculation of c_{s_i} to use an outlier-based system, in which only guesses $s_{\mathsf{Argmax}_\ell(b_{s_\ell}^{(j)})}$ and $s_{\mathsf{Argmin}_\ell(b_{s_\ell}^{(j)})}$ receive contributions to their amplified compressibility score from any round j.

More precisely, we set

$$c_{s_i} = -\sum_{j=1}^{r} R(b_{s_i}^{(j)})$$

where R computes the points awarded for a given round as follows.

$$R(b_{s_i}^{(j)}) = \begin{cases} \mathsf{Min}_{\ell \neq i}(b_{s_\ell}^{(j)}) - b_{s_i}^{(j)} & \text{if } i = \mathsf{Argmin}_\ell(b_{s_\ell}^{(j)}) \\ b_{s_i}^{(j)} - \mathsf{Max}_{\ell \neq i}(b_{s_\ell}^{(j)}) & \text{if } i = \mathsf{Argmax}_\ell(b_{s_\ell}^{(j)}) \\ 0 & \text{otherwise.} \end{cases}$$

That is, the guesses with the largest or smallest $b_{s_i}^{(j)}$ are awarded the difference between their own score and the guess with the second largest or smallest $b_{s_i}^{(j)}$, respectively. To remain consistent with our usual approach for calculating compressibility, we negate all the amplified compressibility scores to ensure that the winning guess still has the smallest score, even though this does result in negative compressibility scores.

Character by character attack evaluation. Prefix length in the character by character attack seems to play a much larger role in Snappy and LZ4 than it did in zlib (see Section VI), with the effects particularly noticeable in the case of Snappy. Because of quirks in the Snappy algorithm, and because Snappy does not compress very aggressively, different prefix lengths converge to very different accuracy values as the number of amplification rounds increases. For example, we achieve 100% accuracy prefixes of length 10, but a maximum of about 90% on prefixes of length 5 and 15. In both the LZ4 and Snappy case, we are significantly more accurate than in zlib for very low numbers of amplifications, likely due to the absence of Huffman encoding. However, since the algorithms sometimes compress less aggressively depending on the length of the string, amplification does not achieve the same level of universal success as in the zlib case. Results of our single character extraction experiments for Snappy and LZ4 appear in Figure 9. This higher likelihood of failure on certain prefix lengths prevents us from achieving very high accuracy with Snappy and LZ4 while extracting characters one at a time.

Figure 10 shows the fraction of bytes extracted in a character by character attack prior to making the first mistake. Since each step of the character by character attack increases the length of the prefix the attacker uses to guess the next character, the attack quickly reaches a point where it must use a prefix of a length on which the next-character extractor performs poorly, resulting in a less effective character by character attack than we saw with zlib.

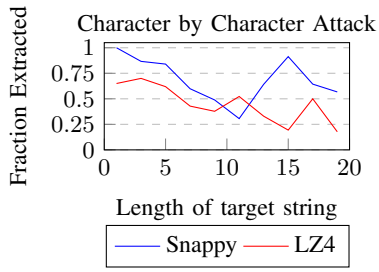

Fig. 10: Average fraction of bytes extracted prior to making the first mistake our using character by character attack on InnoDB, for varying lengths of target strings. Averaged over 10 trials per target string length. Target string does not include known 14-byte prefix.

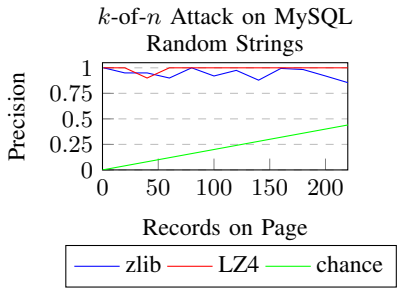

Fig. 11: Precision (fraction of correct guesses) of the k-of-n attack with $n = 500$ and k set to the number of records on the page, while using MySQL with encryption turned off. Snappy is absent because MySQL page compression only supports zlib and LZ4.

Appendix D
Rollback-based attack against InnoDB+MySQL

We explored the extension of our attacks from MariaDB to MySQL, finding that our general method can apply with some caveats. Our attacks succeed with results comparable to MariaDB when compression is turned on without encryption, allowing us to extract table contents without directly reading them from storage. For example, Figure 11 evaluates the k-of-n attack in compression-only mode, demonstrating the potential for a compression side channel in MySQL.

However, when encryption is turned on, MySQL behaves differently from MariaDB in that it does not shrink files on disk as tables become more compressible due to updates. This prevents our attacks from translating as-is to MySQL. We overcome this obstacle by using a stronger threat model where the attacker can, instead of updating rows in the table, *roll back* the table files on the filesystem. While this threat model is stronger than the one described in Section III, it is worth noting that even an attacker with filesystem write capabilities on the database server, which would provide roll back power, would still not be able to read an encrypted database without the proper database permissions.

With this additional power, our attack simulates an update to guess a string by inserting rows, rolling back state to remove the inserted rows, and inserting rows again. The ability to roll back the table state allows us to calculate compressibility scores for multiple guesses, as we can roll back the state instead of re-shrinking the table via updates between each guess. By observing the changes in table size as different guesses are inserted, we can once again execute our attacks on InnoDB in MySQL. We validated this approach with a small-scale experiment where we detect which of a list of names has been inserted into a table.

Appendix E
DBREACH attack using a web frontend

To demonstrate the feasbility of DBREACH when the attacker has access to a frontend web-service rather that directly to the database API, we implemented an API point and a web frontend that uses it store and access data in the database, and instantiated the DBREACH attacks through it. Our web frontend (see Figure 2) takes names as inputs, and stores them in a database. This illustrates a sensitive list of names, such as people signed on a political petition, or people who filled a certain medical form. The frontend invokes a REST API to store the names in an InnoDB database table. We simulate an attacker that has access to the underlying size of the table by adding an API call that provides it. In reality, an attacker could obtain it by monitoring the table size after gaining filesystem read access on the server. The attacker calls the API to perform many insertions of guesses, along with the necessary filler rows; by doing so, we were able to successfully extract a secretly inserted name.

Note that our API does not sanitize the strings. However, we note that sanitization would not prevent the attack, since our attack does not have to use any unusual characters. At best, it would have a small impact on accuracy by preventing the attacker from using rare characters for filler rows.