black hat USA 2021

DBREACH

Database Reconnaissance and Exfiltration via Adaptive Compression Heuristics

Mathew Hogan	Saba Eskandarian	Yan Michalevsky
Stanford University	UNC Chapel Hill	Anjuna Security

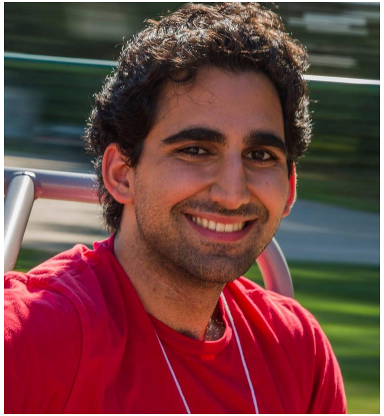
Who We Are

Mathew Hogan

- MS Candidate in CS at Stanford, Security track
- BS in CS from Stanford, Systems track

Yan Michalevsky

- CTO and co-founder at Anjuna.io
- PhD in Security and Crypto from Stanford



Saba Eskandarian

- Assistant Prof. at UNC Chapel Hill
- PhD in Crypto and Security from Stanford

Outline

1. Background
2. Our Attack
3. Roadblocks & Optimizations
4. Analysis
5. Mitigations
6. Conclusion

Background

Encryption Security

Informally,

a ciphertext reveals nothing about the message being encrypted

Encryption Security

Informally,
a ciphertext reveals nothing about the message being encrypted,
except the message’s length.

Encryption Security

Informally,
a ciphertext reveals nothing about the message being encrypted,
except the message's length.

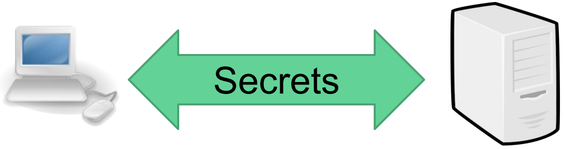
Key idea: use compression to reveal information about the original content

John Kelsey. "Compression and Information Leakage of Plaintext," FSE 2002.

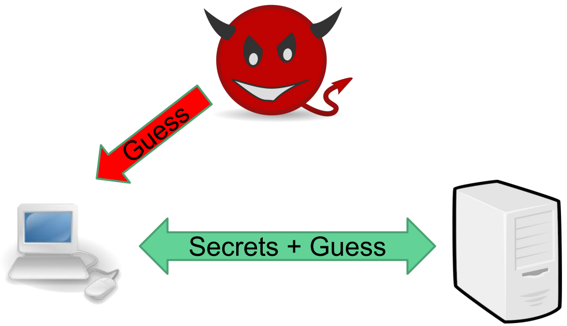
CRIME/BREACH (2012/13)

Secret included in encrypted and compressed messages between client and server

CRIME/BREACH (2012/13)

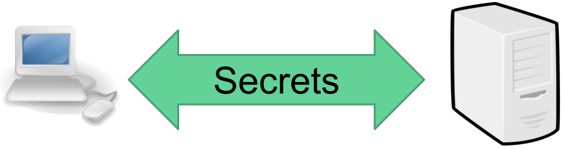


Secret included in encrypted and compressed messages between client and server

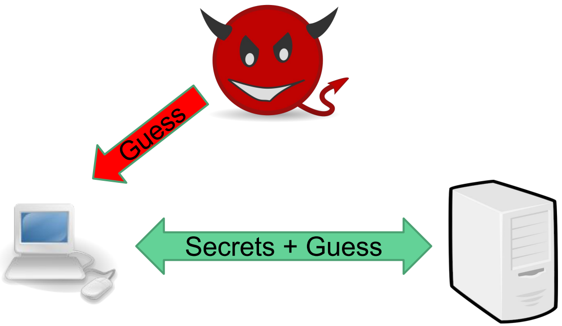


Adversary gets client to include its guess in messages to server (e.g., via malicious Javascript)

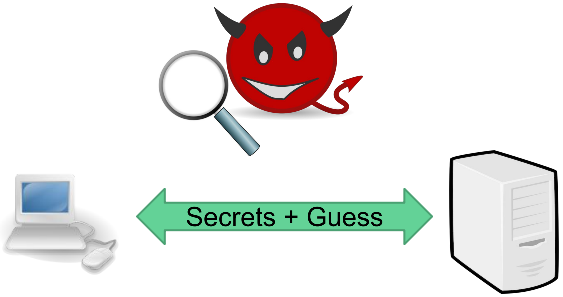
CRIME/BREACH (2012/13)



Secret included in encrypted and compressed messages between client and server

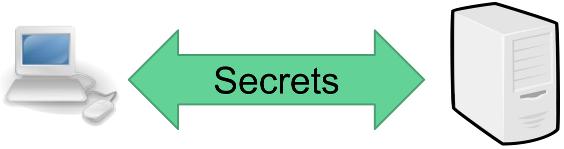


Adversary gets client to include its guess in messages to server (e.g., via malicious Javascript)

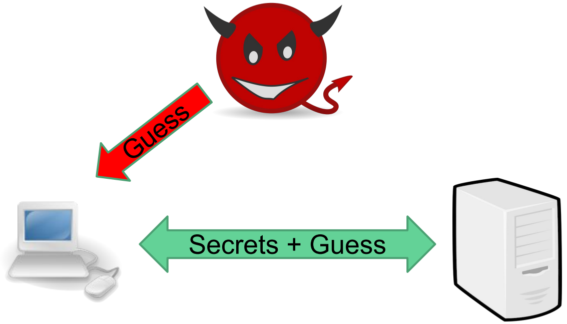


Adversary observes size of encrypted messages to see if guess compresses with secrets

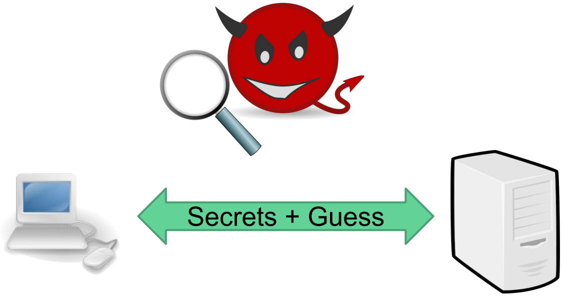
CRIME/BREACH (2012/13)



Secret included in encrypted and compressed messages between client and server



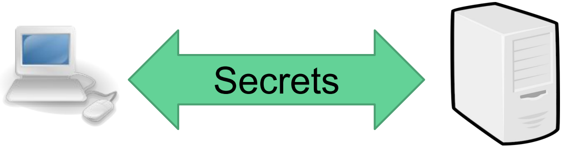
Adversary gets client to include its guess in messages to server (e.g., via malicious Javascript)



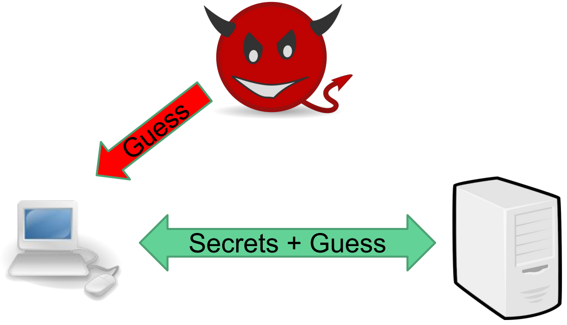
Adversary observes size of encrypted messages to see if guess compresses with secrets

Attack requirement 1:
Encryption + Compression

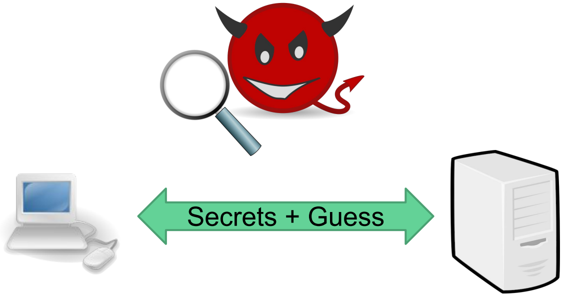
CRIME/BREACH (2012/13)



Secret included in encrypted and compressed messages between client and server



Adversary gets client to include its guess in messages to server (e.g., via malicious Javascript)

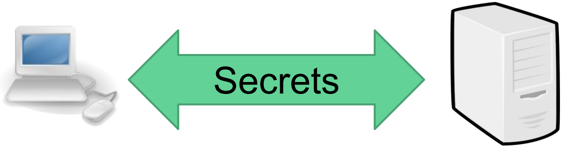


Adversary observes size of encrypted messages to see if guess compresses with secrets

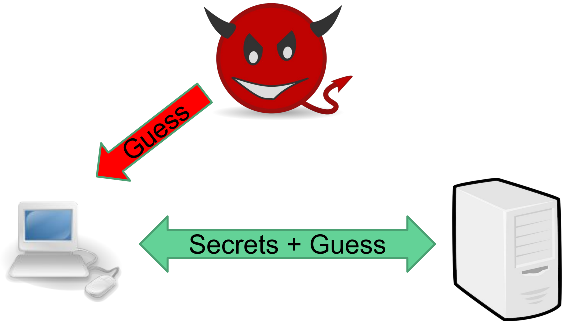
<u>Attack requirement 1:</u>
Encryption + Compression

<u>Attack requirement 2:</u>
Ability to inject messages

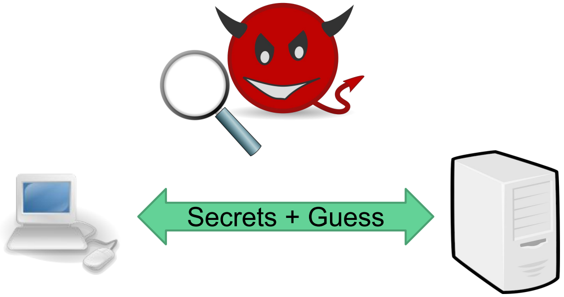
CRIME/BREACH (2012/13)



Secret included in encrypted and compressed messages between client and server



Adversary gets client to include its guess in messages to server (e.g., via malicious Javascript)



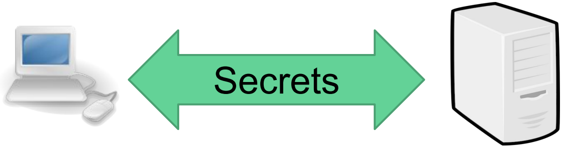
Adversary observes size of encrypted messages to see if guess compresses with secrets

Attack requirement 1:
Encryption + Compression

Attack requirement 2:
Ability to inject messages

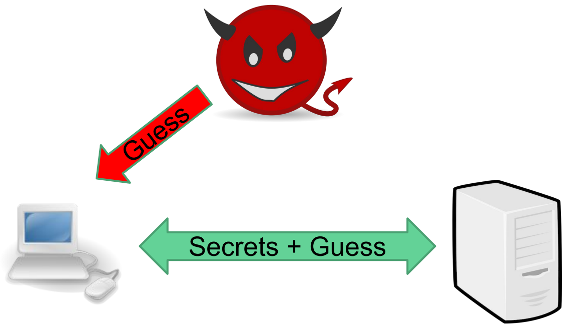
Attack requirement 3:
Access to message size

CRIME/BREACH (2012/13)



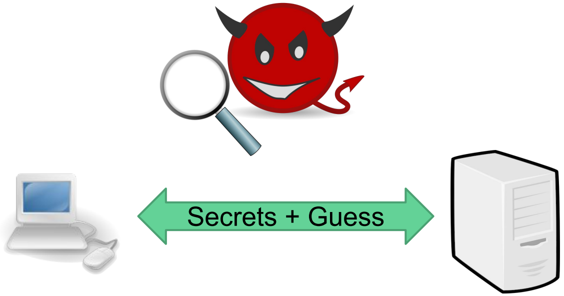
Secret included in encrypted and compressed messages between client and server

Attack requirement 1:
Encryption + Compression



Adversary gets client to include its guess in messages to server (e.g., via malicious Javascript)

Attack requirement 2:
Ability to inject messages



Adversary observes size of encrypted messages to see if guess compresses with secrets

Attack requirement 3:
Access to message size

Where else do all these factors come together?

DBREACH in a Nutshell

Compression side-channel attack against databases

Attacker recovers other users’ encrypted content

Extends techniques from CRIME/BREACH beyond TLS to database context

MariaDB/InnoDB Encryption and Compression

Data-at-Rest Encryption - transparently encrypt data before writing to disk

MariaDB/InnoDB Encryption and Compression

Data-at-Rest Encryption - transparently encrypt data before writing to disk

Compression - many options, compression before encryption

MariaDB/InnoDB Encryption and Compression

Data-at-Rest Encryption - transparently encrypt data before writing to disk

Compression - many options, compression before encryption

- Storage engine independent compression
- InnoDB table compression
- InnoDB transparent page compression

MariaDB/InnoDB Encryption and Compression

Data-at-Rest Encryption - transparently encrypt data before writing to disk

Compression - many options, compression before encryption

- Storage engine independent compression
- InnoDB table compression
- **InnoDB transparent page compression**

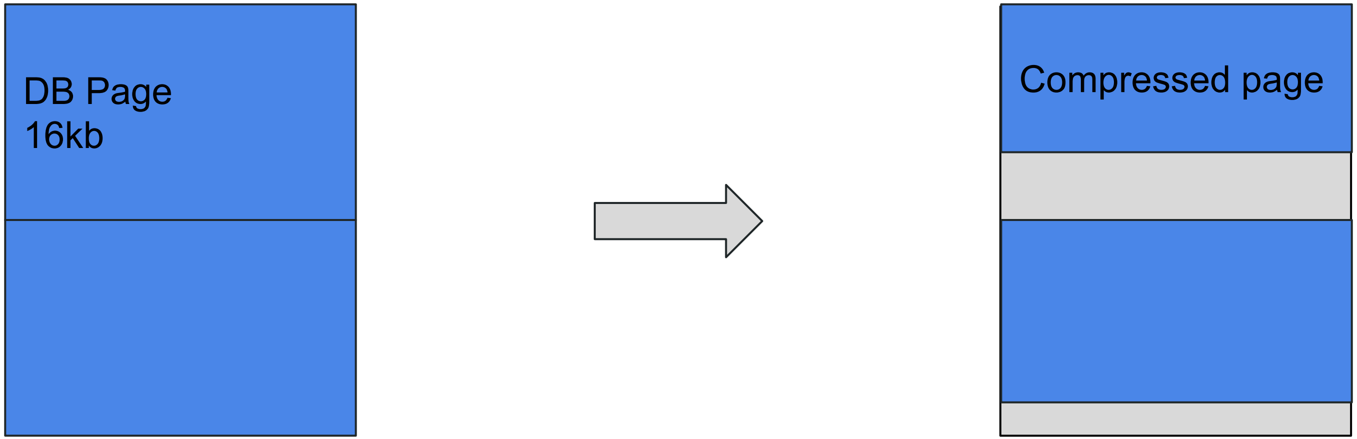
InnoDB Page Compression

Handles compression transparently before encrypting and writing to disk

InnoDB Page Compression

Handles compression transparently before encrypting and writing to disk

Compresses data *within* each database page

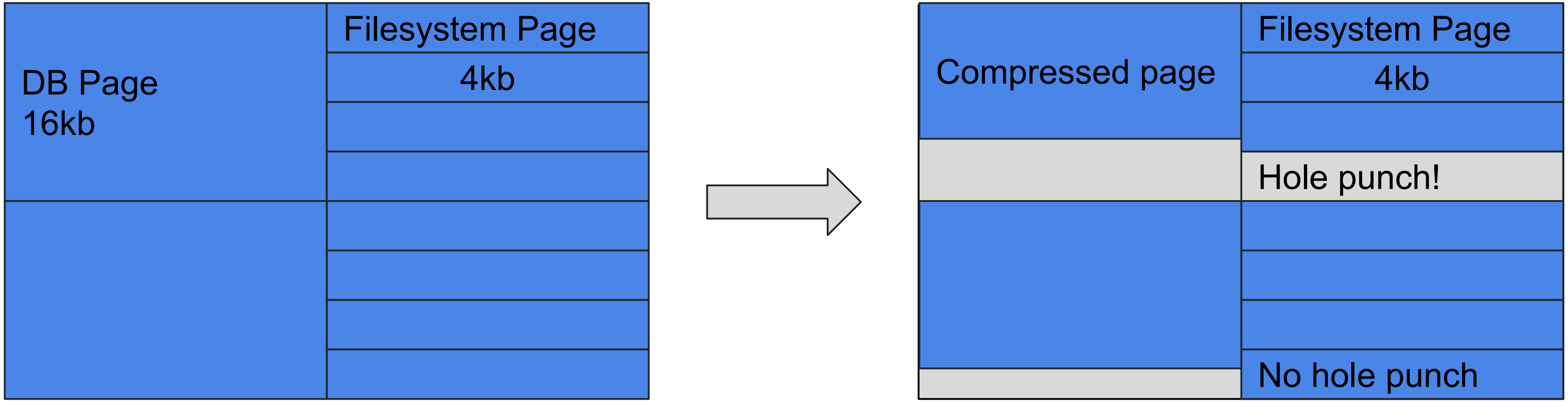


InnoDB Page Compression

Handles compression transparently before encrypting and writing to disk

Compresses data *within* each database page

Uses *hole punching* to save space

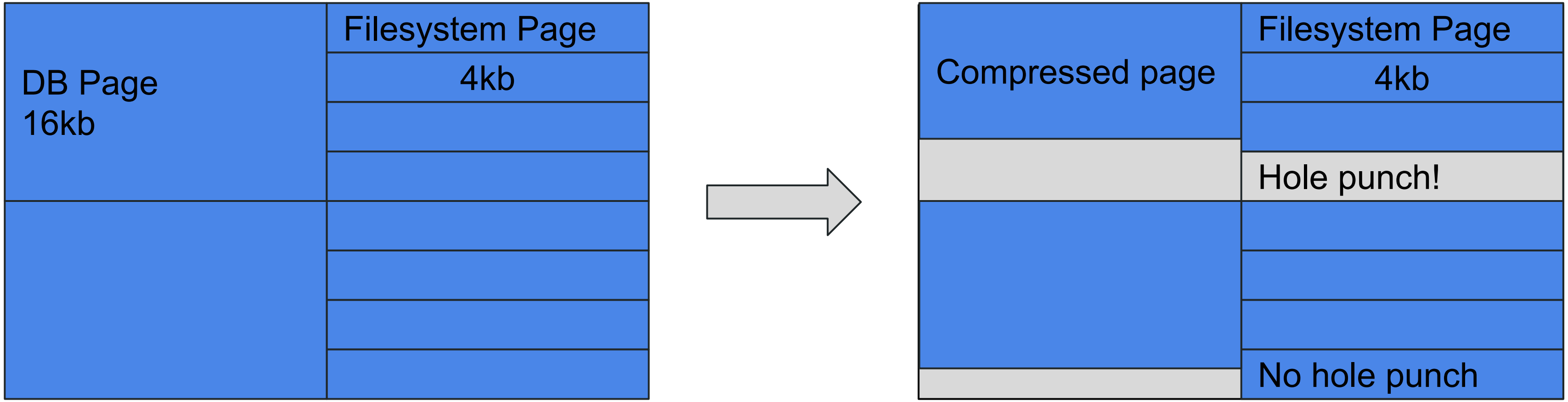


InnoDB Page Compression

Handles compression transparently before encrypting and writing to disk

Compresses data *within* each database page

Uses *hole punching* to save space, only helps when there is enough compression to remove a whole filesystem page



Supported Compression Algorithms

- zlib (default)
- lz4
- lzo
- lzma
- bzip2
- snappy

Supported Compression Algorithms

- **zlib (default)**
- **lz4**
- lzo
- lzma
- bzip2
- **snappy**

Only zlib, lz4, and snappy installed with standard Ubuntu build

Supported Compression Algorithms

- **zlib (default)**
- **lz4**
- lzo
- lzma
- bzip2
- **snappy**

Only zlib, lz4, and snappy installed with standard Ubuntu build

All three begin with LZ77-style sliding window compression

Supported Compression Algorithms

- **zlib (default)**
- **lz4**
- lzo
- lzma
- bzip2
- **snappy**

Only zlib, lz4, and snappy installed with standard Ubuntu build

All three begin with LZ77-style sliding window compression

zlib additionally has a Huffman Coding step

Our Attack

Threat Model

An attacker needs the ability to:

- Insert and update into a database table
- Assess the size of the compressed table

Threat Model

An attacker needs the ability to:

- Insert and update into a database table
    - Can be achieved through a web interface
    - Column-level permissions grant update ability if user has partial SELECT ability
- Assess the size of the compressed table

Threat Model

An attacker needs the ability to:

- Insert and update into a database table
    - Can be achieved through a web interface
    - Column-level permissions grant update ability if user has partial SELECT ability
- Assess the size of the compressed table
    - Can read size of table file by gaining read access to the filesystem

Threat Model

An attacker needs the ability to:

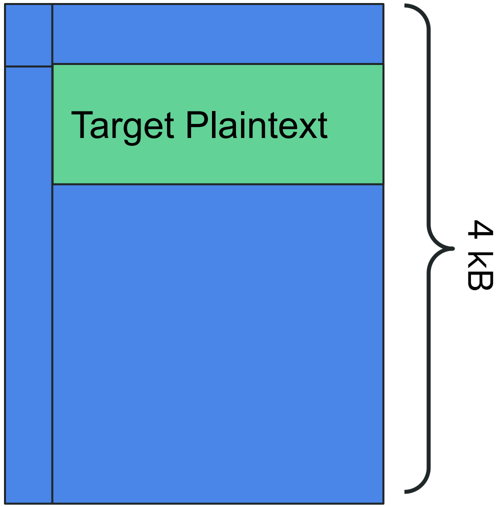
- Insert and update into a database table
  - Can be achieved through a web interface
  - Column-level permissions grant update ability if user has partial SELECT ability.
- Assess the size of the compressed table
  - Can read size of table file by gaining read access to filesystem

If UPDATE permissions can’t be achieved, an attacker with write access can force an update by rolling back the table file and inserting.

Attack Algorithm

Attack Algorithm

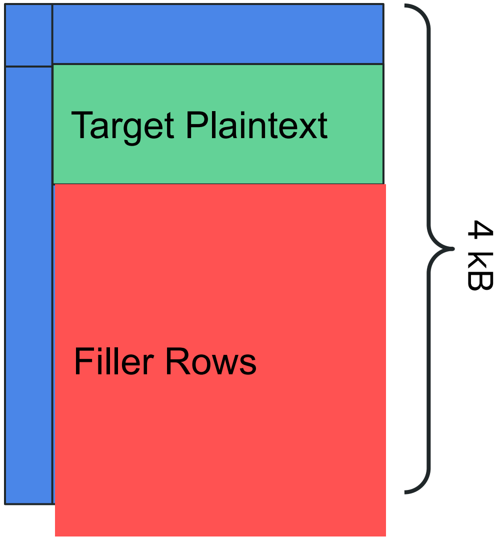
Table Layout



Attack Algorithm

1. Add “filler rows” until table grows

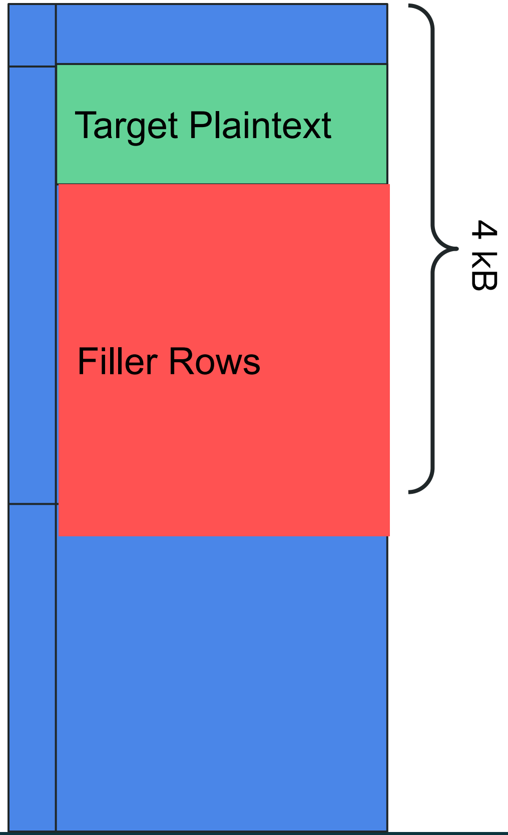
Table Layout



Attack Algorithm

1. Add “filler rows” until table grows

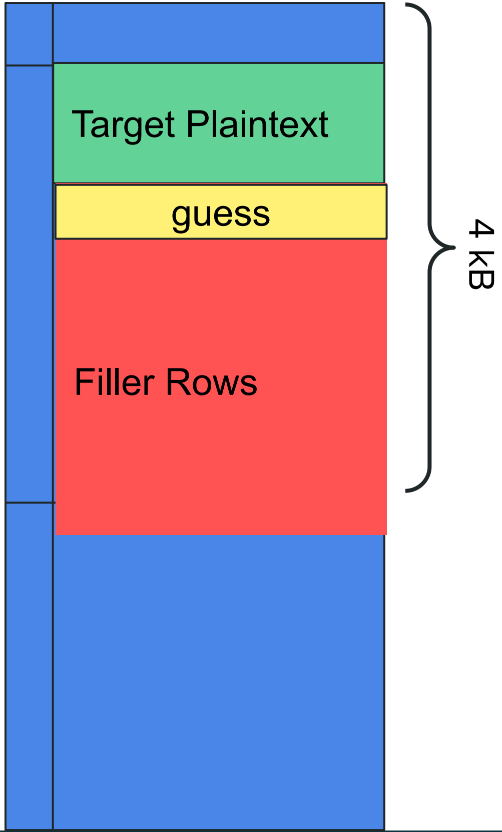
Table Layout



Attack Algorithm

1. Add “filler rows” until table grows
2. Update first filler row to contain guess

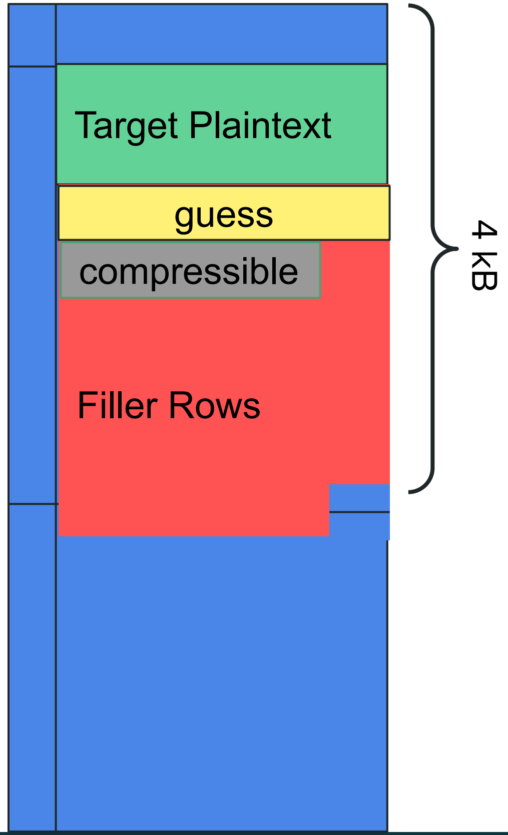
Table Layout



Attack Algorithm

1. Add “filler rows” until table grows
2. Update first filler row to contain guess
3. Byte by byte, make filler rows compressible

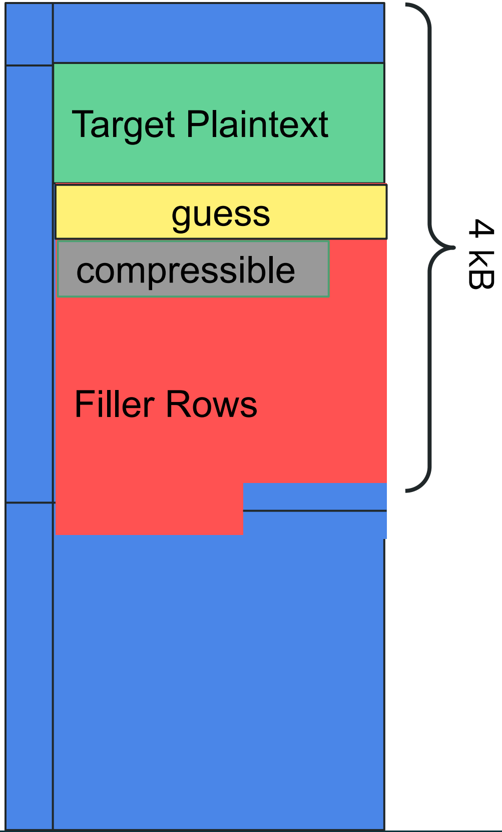
Table Layout



Attack Algorithm

1. Add “filler rows” until table grows
2. Update first filler row to contain guess
3. Byte by byte, make filler rows compressible

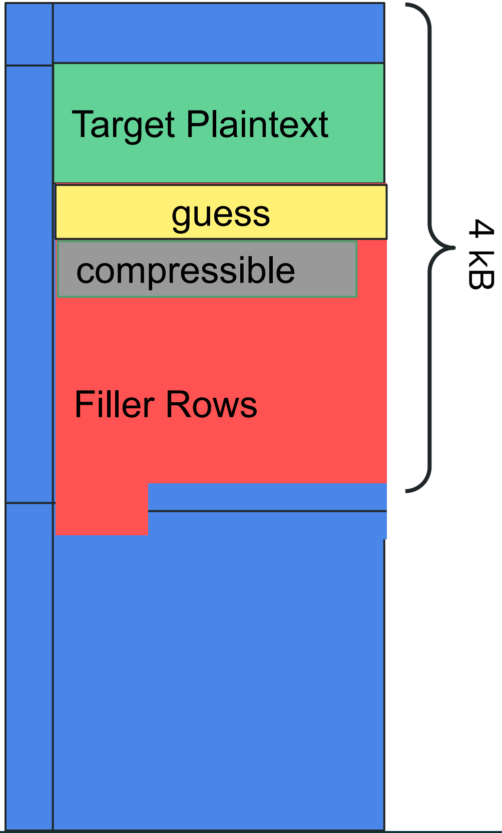
Table Layout



Attack Algorithm

1. Add “filler rows” until table grows
2. Update first filler row to contain guess
3. Byte by byte, make filler rows compressible

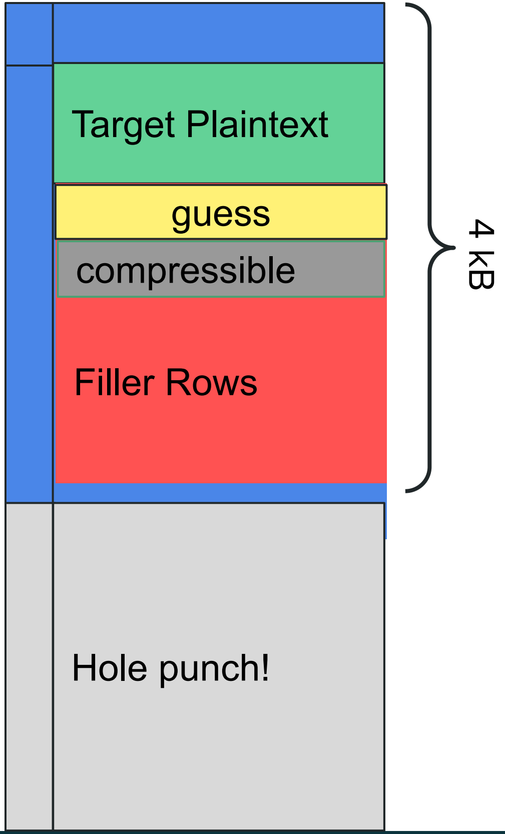
Table Layout



Attack Algorithm

1. Add “filler rows” until table grows
2. Update first filler row to contain guess
3. Byte by byte, make filler rows compressible

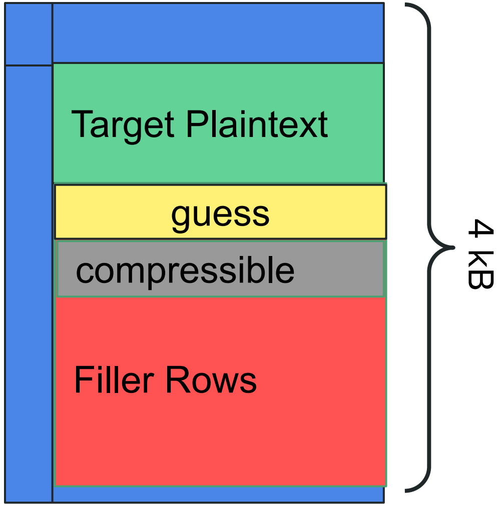
Table Layout



Attack Algorithm

1. Add "filler rows" until table grows
2. Update first filler row to contain guess
3. Byte by byte, make filler rows compressible
   a. The number of bytes until the table shrinks determines this guess's "compressibility score"

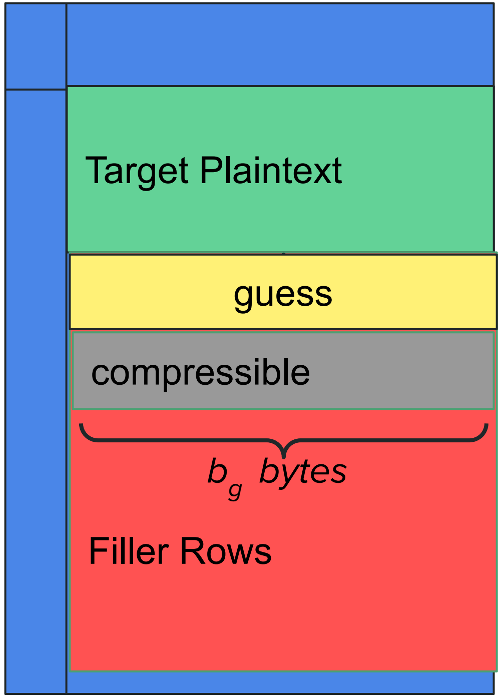
Table Layout



Compressibility Scores

Compressibility Scores

For a guess g, let b_g be the number of bytes that we made compressible in order to shrink the table.

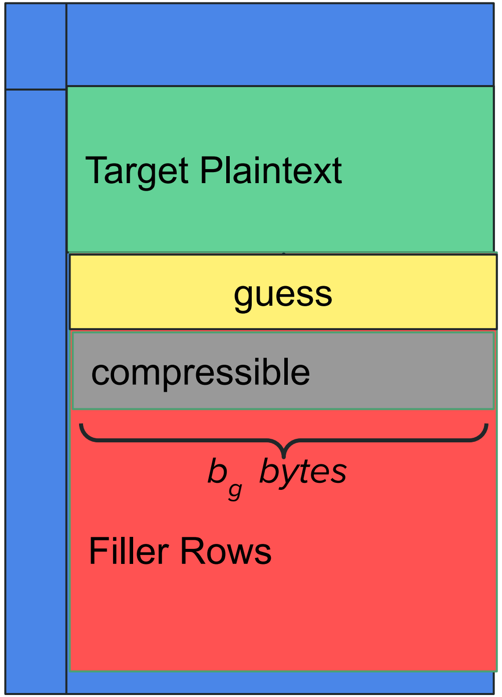


Compressibility Scores

For a guess *g*, let b_g be the number of bytes that we made compressible in order to shrink the table.

The compressibility score c_g is calculated as follows:

$$\boldsymbol{c_g = 1 / b_g}$$



Attack Varieties

- *K-of-n* attack: Given *n* options, which *k* are most likely to be in the table?

Attack Varieties

- *K-of-n* attack: Given *n* options, which *k* are most likely to be in the table?
    - Simply pick the *k* guesses with the highest compressibility scores!

Attack Varieties

- *K-of-n* attack: Given *n* options, which *k* are most likely to be in the table?
  - Simply pick the *k* guesses with the highest compressibility scores!
- Decision attack: Is a guess in the table?

Attack Varieties

- *K-of-n* attack: Given *n* options, which *k* are most likely to be in the table?
  - Simply pick the *k* guesses with the highest compressibility scores!
- Decision attack: Is a guess in the table?
  - More complicated
  - We need a reference point for compressibility scores

Decision Attack: Is a guess in the table?

To gain reference points, we calculate scores $\boldsymbol{s}_{\boldsymbol{yes}}$ and $\boldsymbol{s}_{\boldsymbol{no}}$ for strings we know to be in and not in the table, respectively.

Decision Attack: Is a guess in the table?

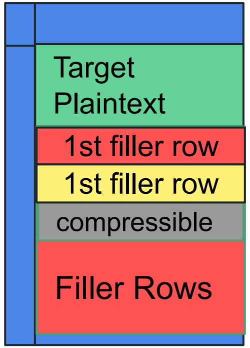
To gain reference points, we calculate scores $\boldsymbol{s_{yes}}$ and $\boldsymbol{s_{no}}$ for strings we know to be in and not in the table, respectively.

- ***Determining s_{yes}***
- ***Determining s_{no}***

Decision Attack: Is a guess in the table?

To gain reference points, we calculate scores $\boldsymbol{s}_{\boldsymbol{yes}}$ and $\boldsymbol{s}_{\boldsymbol{no}}$ for strings we know to be in and not in the table, respectively.

- ***Determining*** $\boldsymbol{s}_{\boldsymbol{yes}}$
  - Use a substring of the first filler row of the same length as the guess
  - Insert substring into the second filler row & determine compressibility

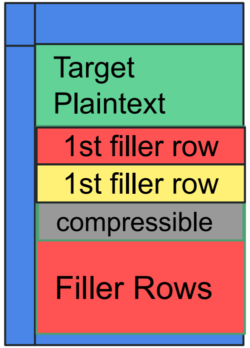


- ***Determining*** $\boldsymbol{s}_{\boldsymbol{no}}$

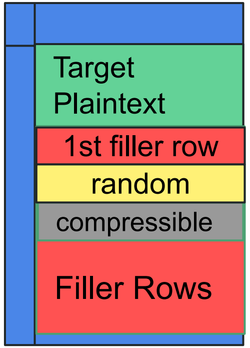
Decision Attack: Is a guess in the table?

To gain reference points, we calculate scores $\boldsymbol{s}_{\boldsymbol{yes}}$ and $\boldsymbol{s}_{\boldsymbol{no}}$ for strings we know to be in and not in the table, respectively.

- ***Determining*** $\boldsymbol{s}_{\boldsymbol{yes}}$
  - Use a substring of the first filler row of the same length as the guess
  - Insert substring into the second filler row & determine compressibility



- ***Determining*** $\boldsymbol{s}_{\boldsymbol{no}}$
  - Use a random string of the same length as the guess
  - Insert string into the second filler row & determine compressibility



Decision Attack: Is a guess in the table?

To gain reference points, we calculate scores $\boldsymbol{s}_{\boldsymbol{yes}}$ and $\boldsymbol{s}_{\boldsymbol{no}}$ for strings we know to be in and not in the table, respectively.

- ***Determining s_{yes}***
  - Use a substring of the first filler row of the same length as the guess
  - Insert substring into the second filler row & determine compressibility

- ***Determining s_{no}***
  - Use a random string of the same length as the guess
  - Insert string into the second filler row & determine compressibility

If a guess's score is within some threshold of $\boldsymbol{s}_{\boldsymbol{yes}}$, answer "yes"

Character-by-Character Extraction

- We can use *1*-of-*n* attack as a subroutine to extract plaintext, char by char:

Character-by-Character Extraction

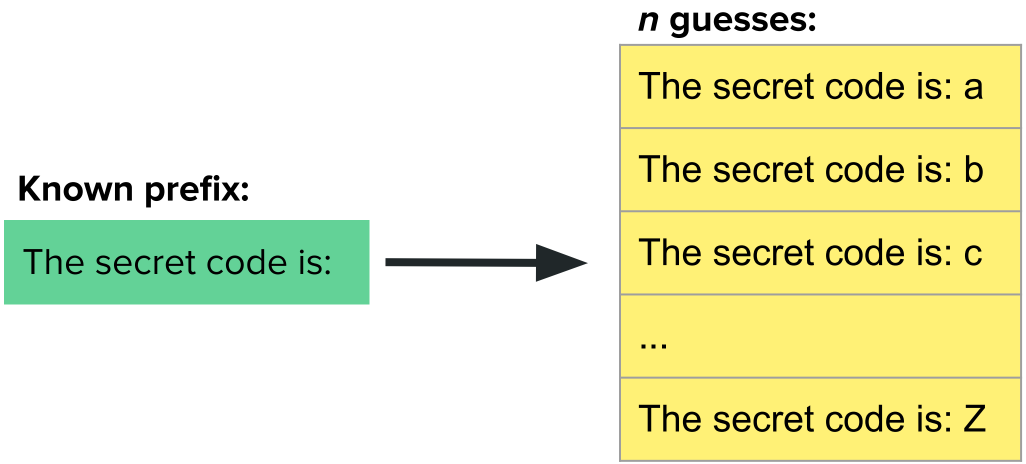
- We can use *1*-of-*n* attack as a subroutine to extract plaintext, char by char:

Known prefix:

The secret code is:

Character-by-Character Extraction

- We can use *1*-of-*n* attack as a subroutine to extract plaintext, char by char:



Known prefix:

The secret code is:

***n* guesses:**

The secret code is: a

The secret code is: b

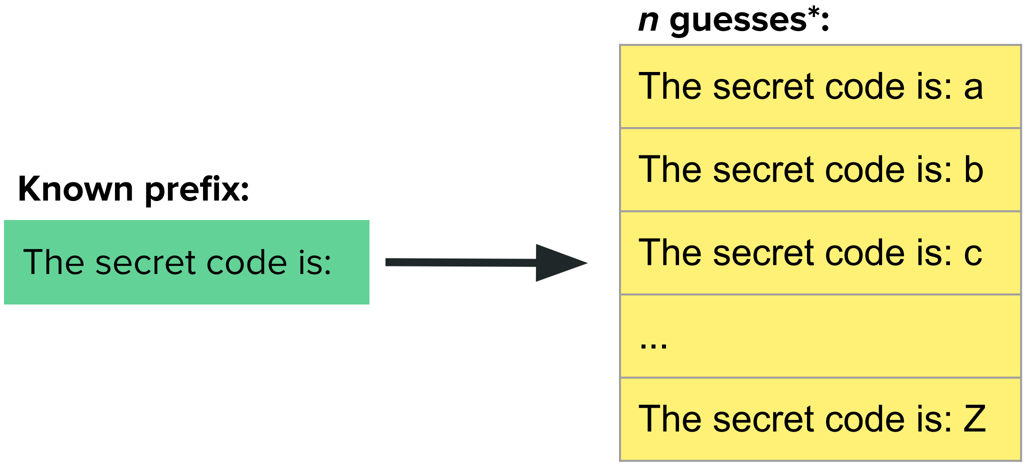
The secret code is: c

...

The secret code is: Z

Character-by-Character Extraction

- We can use *1-of-n* attack as a subroutine to extract plaintext, char by char:



Known prefix:

The secret code is:

***n* guesses*:**

The secret code is: a

The secret code is: b

The secret code is: c

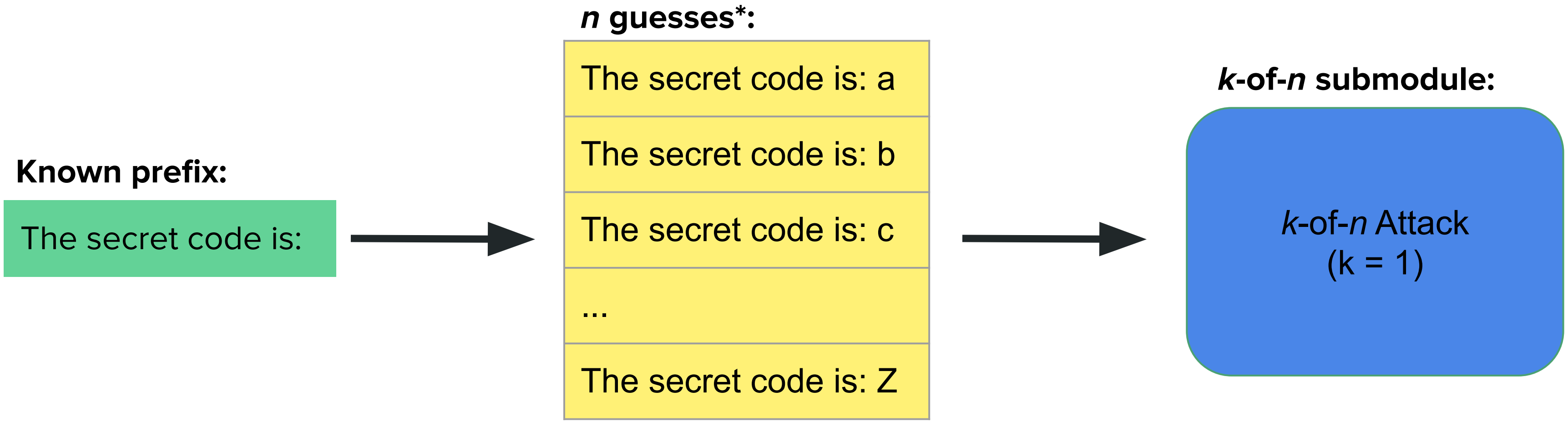
...

The secret code is: Z

*$n = |\Sigma|$, where Σ is the alphabet of all possible characters

Character-by-Character Extraction

- We can use *1*-of-*n* attack as a subroutine to extract plaintext, char by char:



Known prefix:

The secret code is:

n **guesses*:**

The secret code is: a

The secret code is: b

The secret code is: c

...

The secret code is: Z

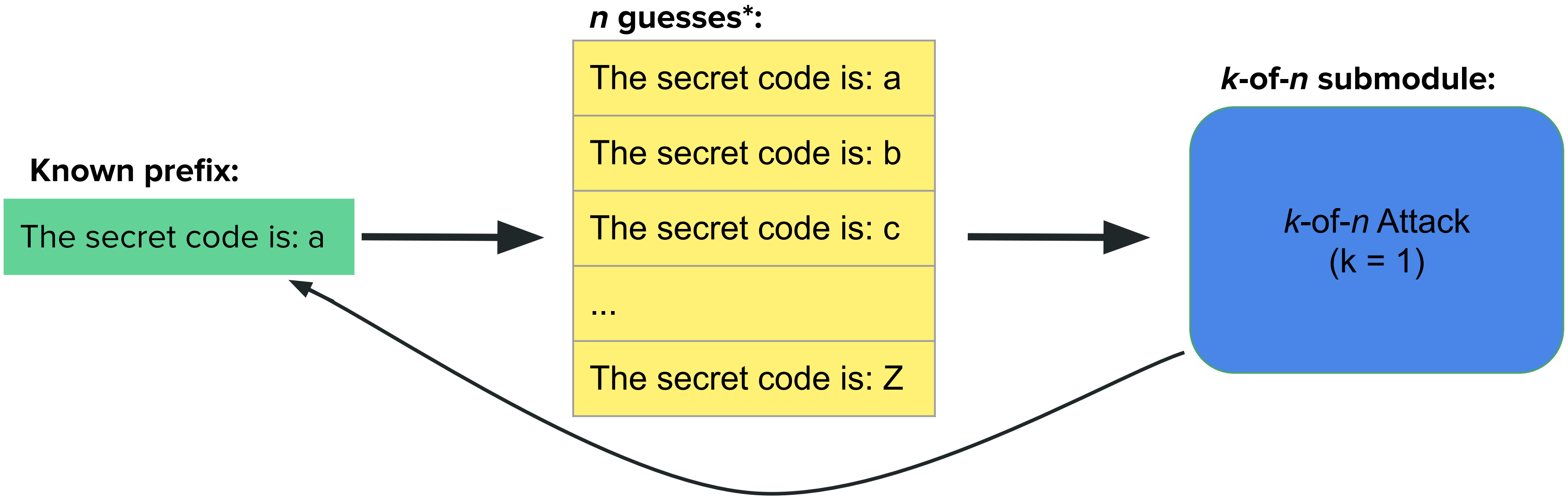
k**-of-*****n*** **submodule:**

k-of-*n* Attack
(k = 1)

*n = |Σ|, where Σ is the alphabet of all possible characters

Character-by-Character Extraction

- We can use *1*-of-*n* attack as a subroutine to extract plaintext, char by char:



Known prefix:

The secret code is: a

***n* guesses*:**

The secret code is: a

The secret code is: b

The secret code is: c

...

The secret code is: Z

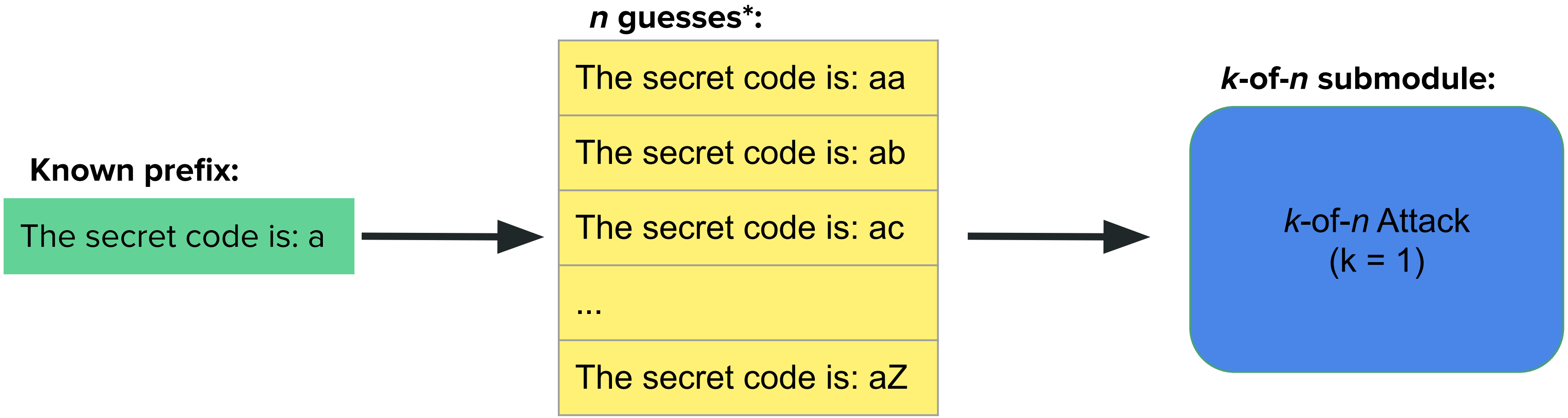
***k*-of-*n* submodule:**

k-of-*n* Attack
(k = 1)

*$n = |\Sigma|$, where Σ is the alphabet of all possible characters

Character-by-Character Extraction

- We can use *1*-of-*n* attack as a subroutine to extract plaintext, char by char:



Known prefix:

The secret code is: a

***n* guesses*:**

The secret code is: aa

The secret code is: ab

The secret code is: ac

...

The secret code is: aZ

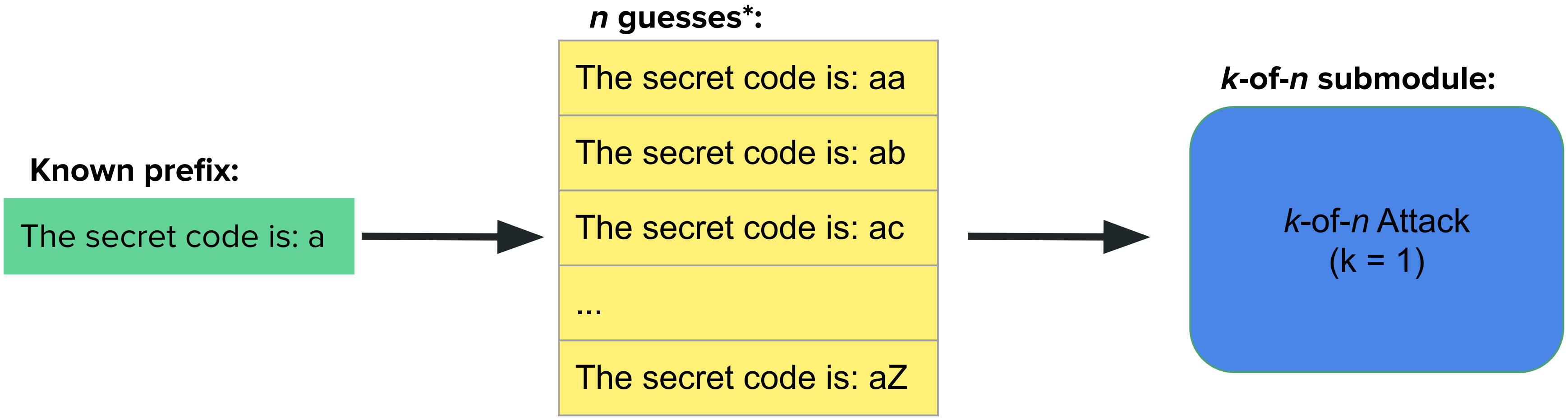
***k*-of-*n* submodule:**

k-of-*n* Attack
(k = 1)

**n* = |Σ|, where Σ is the alphabet of all possible characters

Character-by-Character Extraction

- We can use *1*-of-*n* attack as a subroutine to extract plaintext, char by char:



Known prefix:

The secret code is: a

***n* guesses*:**

The secret code is: aa

The secret code is: ab

The secret code is: ac

...

The secret code is: aZ

***k*-of-*n* submodule:**

k-of-*n* Attack
(k = 1)

**n* = |Σ|, where Σ is the alphabet of all possible characters

Roadblocks & Optimizations

Substring/Superstring Problem

Might get false positives if guess is a:

- **Substring**: A guess is a substring of what is in the table

- **Superstring**: What is in the table is a substring of the guess

Substring/Superstring Problem

Might get false positives if guess is a:

- **Substring**: A guess is a substring of what is in the table

plaintext:	“ground truth”
guess:	“ground”

- **Superstring**: What is in the table is a substring of the guess

Substring/Superstring Problem

Might get false positives if guess is a:

- **Substring**: A guess is a substring of what is in the table

plaintext:	“ground truth”
guess:	“ground”

- **Superstring**: What is in the table is a substring of the guess

plaintext:	“ground truth”
guess:	“ground truth regarding…”

Substring/Superstring Problem

Might get false positives if guess is a:

- **Substring**: A guess is a substring of what is in the table

plaintext:	“ground”
guess:	“ground”

👍

- **Superstring**: What is in the table is a substring of the guess

plaintext:	“ground truth”
guess:	“ground truth regarding…”

Substring/Superstring Problem

Might get false positives if guess is a:

- **Substring**: A guess is a substring of what is in the table

plaintext:	“ground”	👍
guess:	“ground”	

- **Superstring**: What is in the table is a substring of the guess

plaintext:	“ground truth”	👎
guess:	“ground truth regarding…”	

Addressing the Superstring Problem

To avoid the superstring problem, we switch to using the $\boldsymbol{s_{yes}}$ / $\boldsymbol{s_{no}}$ reference point strategy on all guesses.

Addressing the Superstring Problem

To avoid the superstring problem, we switch to using the $\boldsymbol{s_{yes}}$ / $\boldsymbol{s_{no}}$ reference point strategy on all guesses.

- A guess’s score is determined by how close its score is to $\boldsymbol{s_{yes}}$ relative to $\boldsymbol{s_{no}}$

Addressing the Superstring Problem

To avoid the superstring problem, we switch to using the $\boldsymbol{s_{yes}}$ / $\boldsymbol{s_{no}}$ reference point strategy on all guesses.

- A guess's score is determined by how close its score is to $\boldsymbol{s_{yes}}$ relative to $\boldsymbol{s_{no}}$
- Since $\boldsymbol{s_{yes}}$ is calculated using a string of the same length as the guess, superstring guesses will only appear partially compressible relative to an exact match.

Addressing the Superstring Problem

To avoid the superstring problem, we switch to using the $\boldsymbol{s_{yes}}$ / $\boldsymbol{s_{no}}$ reference point strategy on all guesses.

- A guess's score is determined by how close its score is to $\boldsymbol{s_{yes}}$ relative to $\boldsymbol{s_{no}}$
- Since $\boldsymbol{s_{yes}}$ is calculated using a string of the same length as the guess, superstring guesses will only appear partially compressible relative to an exact match.

Still vulnerable to false positives if the superstring is not much longer than the ground truth (recall that we only have to be close to $\boldsymbol{s_{yes}}$ and not precisely match it).

Overcoming Noise in the Side Channel

There are multiple sources of noise in the compression side-channel that can lead to false positives or negatives:

Overcoming Noise in the Side Channel

There are multiple sources of noise in the compression side-channel that can lead to false positives or negatives:

- Huffman encoding

Overcoming Noise in the Side Channel

There are multiple sources of noise in the compression side-channel that can lead to false positives or negatives:

- Huffman encoding
  - Typically does not overwhelm signal in *k*-of-*n* or decision attacks
  - Poses problem for char-by-char extraction attack

Overcoming Noise in the Side Channel

There are multiple sources of noise in the compression side-channel that can lead to false positives or negatives:

- Huffman encoding
  - Typically does not overwhelm signal in *k*-of-*n* or decision attacks
  - Poses problem for char-by-char extraction attack
- Compression with irrelevant parts of the tablespace file

Overcoming Noise in the Side Channel

There are multiple sources of noise in the compression side-channel that can lead to false positives or negatives:

- Huffman encoding
  - Typically does not overwhelm signal in *k*-of-*n* or decision attacks
  - Poses problem for char-by-char extraction attack
- Compression with irrelevant parts of the tablespace file
  - **Solution:** Randomly choose filler data from disjoint alphabet

Overcoming Noise in the Side Channel

There are multiple sources of noise in the compression side-channel that can lead to false positives or negatives:

- Huffman encoding
    - Typically does not overwhelm signal in *k*-of-*n* or decision attacks
    - Poses problem for char-by-char extraction attack
- Compression with irrelevant parts of the tablespace file
    - **Solution:** Randomly choose filler data from disjoint alphabet
- Compression within the guess itself

Overcoming Noise in the Side Channel

There are multiple sources of noise in the compression side-channel that can lead to false positives or negatives:

- Huffman encoding
  - Typically does not overwhelm signal in *k*-of-*n* or decision attacks
  - Poses problem for char-by-char extraction attack
- Compression with irrelevant parts of the tablespace file
  - **Solution:** Randomly choose filler data from disjoint alphabet
- Compression within the guess itself
  - **Solution:** Penalize internally compressible guesses

Overcoming Noise in the Side Channel

There are multiple sources of noise in the compression side-channel that can lead to false positives or negatives:

- Huffman encoding
    - Typically does not overwhelm signal in *k*-of-*n* or decision attacks
    - Poses problem for char-by-char extraction attack
- Compression with irrelevant parts of the tablespace file
    - **Solution:** Randomly choose filler data from disjoint alphabet
- Compression within the guess itself
    - **Solution:** Penalize internally compressible guesses
- Fragmented tablespace ➔ insertion onto different pages

Overcoming Noise in the Side Channel

There are multiple sources of noise in the compression side-channel that can lead to false positives or negatives:

- Huffman encoding
  - Typically does not overwhelm signal in *k*-of-*n* or decision attacks
  - Poses problem for char-by-char extraction attack
- Compression with irrelevant parts of the tablespace file
  - **Solution:** Randomly choose filler data from disjoint alphabet
- Compression within the guess itself
  - **Solution:** Penalize internally compressible guesses
- Fragmented tablespace ➡ insertion onto different pages
  - **Solution:** Detect & retry

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
    - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
    - Takes only $\log_2 R$ updates per guess

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
    - If R is the max row size, never takes more than R compressible bytes to compress a table
    - Takes only $\log_2 R$ updates per guess

Before:

Filler Row

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

Before:

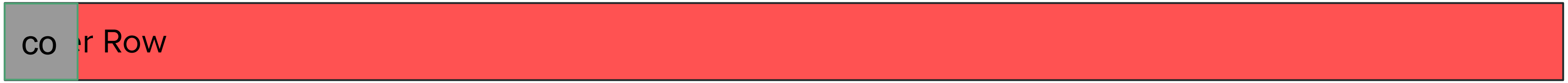
c ller Row

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

Before:

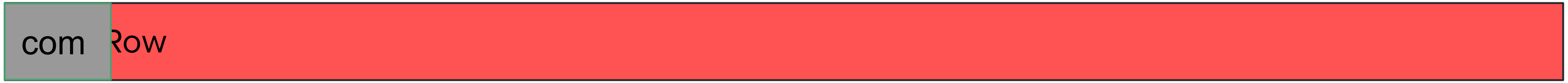


Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

Before:



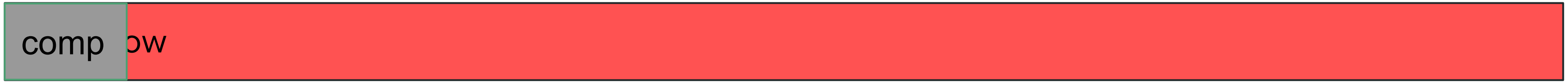
com Row

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

Before:



comp ow

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

Before:

compr

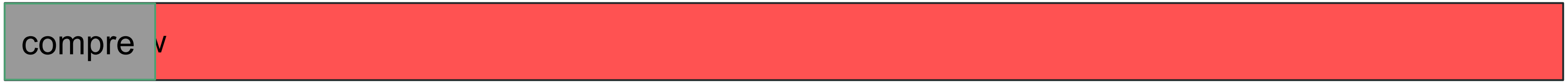
w

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

Before:



Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

Before:

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

Before:

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
    - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
    - Takes only $\log_2 R$ updates per guess

Before:

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
    - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
    - Takes only $\log_2 R$ updates per guess

Before:

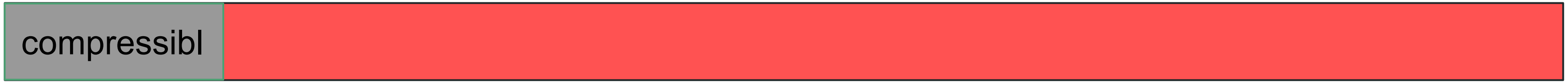
compressib

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
    - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
    - Takes only $\log_2 R$ updates per guess

Before:

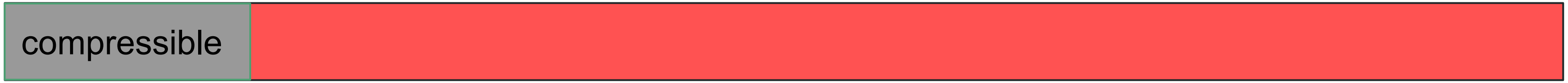


Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

Before:



Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

After:

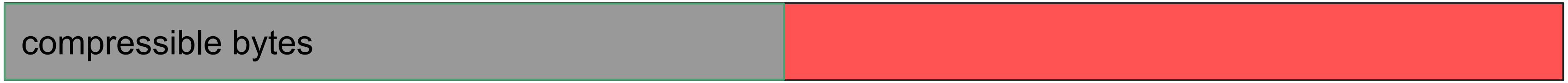
Filler Row

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

After:



Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

After:

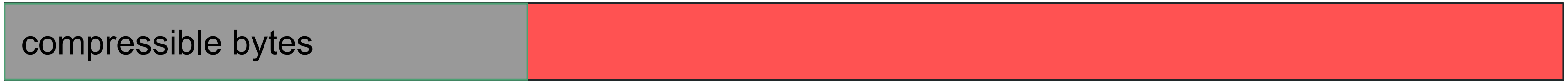
compressible bytes

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

After:



Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
    - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
    - Takes only $\log_2 R$ updates per guess

After:

compressible bytes

Maximizing Efficiency

To maximize efficiency of the attack and evade detection, we want to perform as few updates and inserts as possible.

- Perform a **binary search** to find the cutoff point of compressible bytes needed to shrink the table, instead of going byte-by-byte
  - If *R* is the max row size, never takes more than *R* compressible bytes to compress a table
  - Takes only $\log_2 R$ updates per guess

After:

compressible bytes

Analysis

Efficiency & Speed

After our binary search optimization, the attack becomes very efficient:

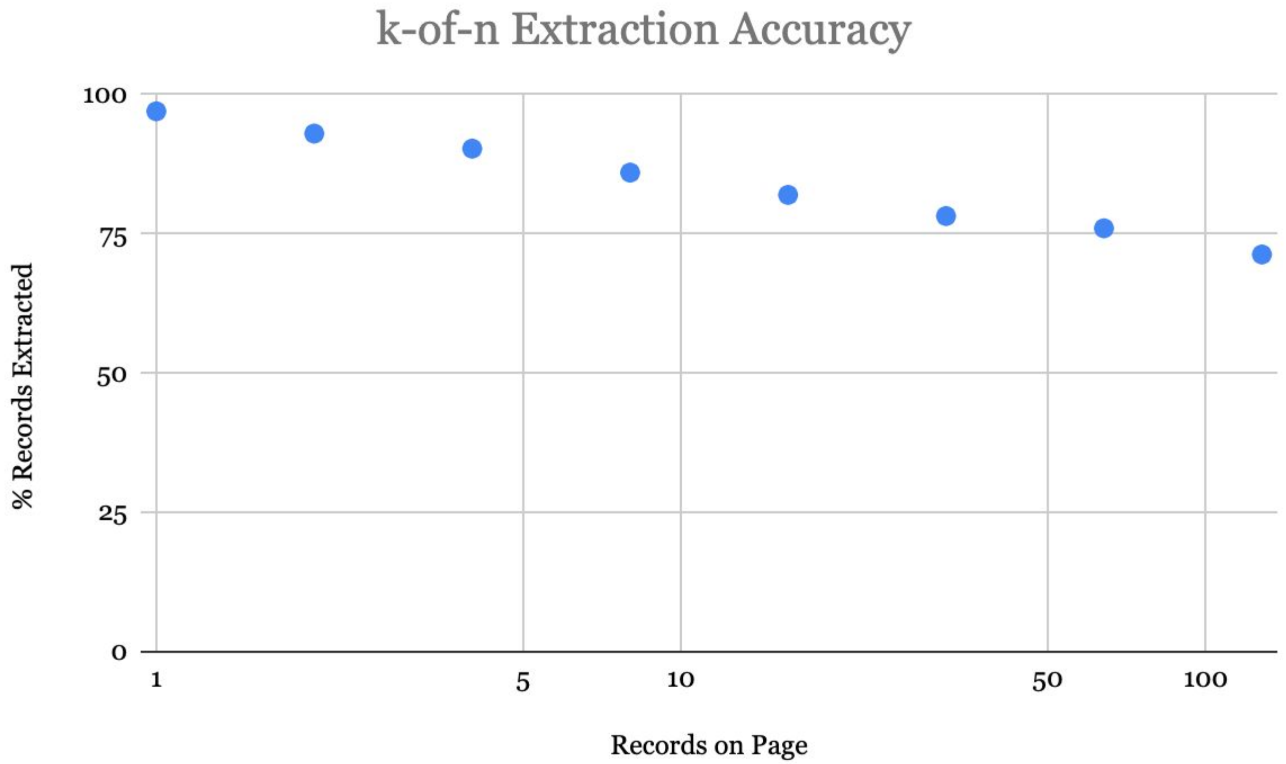
Let *R* be the maximum size of a row

- Insertion of filler rows
  - We must initially insert at most *page_size / R* rows to fill up the page
  - In practice, with an empty page and *R = 200*, this takes about 30 insertions
- Updates per guess
  - $\log_2 R$ updates per guess

Thus, for *n* guesses we perform $O(R + n \log R)$ database actions.

In practice, with *R = 200*, a single guess took 0.2-0.4 seconds.

Accuracy


k-of-n Extraction Accuracy
100
75
50
25
0
% Records Extracted
1
5
10
50
100
Records on Page

Vulnerability of Other Systems

Nothing specific about MariaDB/InnoDB’s implementation makes them vulnerable.

Vulnerability of Other Systems

Nothing specific about MariaDB/InnoDB’s implementation makes them vulnerable.

The issue is much more fundamental:

The DB compresses attacker and victim data together.

Vulnerability of Other Systems

Nothing specific about MariaDB/InnoDB’s implementation makes them vulnerable.

The issue is much more fundamental:

The DB compresses attacker and victim data together.

We believe that other RDBMSs and storage engines are vulnerable to the same attack. MySQL is especially likely to be vulnerable.

Mitigations

Prevention

Recommendations for database administrators & developers using databases:

Prevention

Recommendations for database administrators & developers using databases:

- **DO NOT** use column-level permissions for SELECT capabilities.

Prevention

Recommendations for database administrators & developers using databases:

- **DO NOT** use column-level permissions for SELECT capabilities.
  - This attack clearly shows that these are broken and provide attackers with half the necessary capabilities.

Prevention

Recommendations for database administrators & developers using databases:

- **DO NOT** use column-level permissions for SELECT capabilities.
  - This attack clearly shows that these are broken and provide attackers with half the necessary capabilities.
- Monitor database usage patterns for unusual activity

Prevention

Recommendations for database administrators & developers using databases:

- **DO NOT** use column-level permissions for SELECT capabilities.
  - This attack clearly shows that these are broken and provide attackers with half the necessary capabilities.
- Monitor database usage patterns for unusual activity
  - Similar to DOS detection: is a single user performing an unusually high number of inserts/updates?

Prevention

Recommendations for database administrators & developers using databases:

- **DO NOT** use column-level permissions for SELECT capabilities.
  - This attack clearly shows that these are broken and provide attackers with half the necessary capabilities.
- Monitor database usage patterns for unusual activity
  - Similar to DOS detection: is a single user performing an unusually high number of inserts/updates?

Only foolproof solution: **Turn off compression.**

Patching the Vulnerability

Recommendations for database developers:

Patching the Vulnerability

Recommendations for database developers:

- **Deprecate column-level permissions** for SELECT.
  - At least until a more comprehensive solution is found.
  - Alternatively, require SELECT permissions on all columns in order to UPDATE.

Patching the Vulnerability

Recommendations for database developers:

- **Deprecate column-level permissions** for SELECT.
  - At least until a more comprehensive solution is found.
  - Alternatively, require SELECT permissions on all columns in order to UPDATE.
- Compress only within rows

Patching the Vulnerability

Recommendations for database developers:

- **Deprecate column-level permissions** for SELECT.
  - At least until a more comprehensive solution is found.
  - Alternatively, require SELECT permissions on all columns in order to UPDATE.
- Compress only within rows
- Or, compress only within rows inserted by the same user / user group

Demo!

DBREACH

- Attack on compression & encryption in databases
- Simple threat model
- **Efficient** and **accurate**

Contact

mhogan1@cs.stanford.edu
yanm2@cs.stanford.edu
saba@cs.unc.edu