DORY: An Encrypted Search System with Distributed Trust

Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica
UC Berkeley

OSDI 2020
End-to-end encrypted filesystems

End-to-end encrypted systems are increasingly popular.
End-to-end encrypted filesystems

End-to-end encrypted systems are increasingly popular.

Provide strong security guarantees if attacker compromises server.
Users expect the ability to search
Users expect the ability to search

Apple
Users expect the ability to search

Apple

Doc 1
Doc 7
Doc 21
Doc 53
Search for end-to-end encrypted filesystems

**Challenge**: server cannot decrypt data to search.

Find all documents with “apple”  

\[ \text{Enc}(\mathcal{P}, \text{doc}_1) \]  
\[ \text{Enc}(\mathcal{P}, \text{doc}_2) \]  
[\ldots]  
\[ \text{Enc}(\mathcal{P}, \text{doc}_n) \]
Tradeoff between security and performance

ORAM-based solutions
[GO96], PathORAM, ...

Searchable Encryption (SE)
[SWP00], [Goh03], [CGKO11], [KPR12], [KP13], [CJJJ+14], [SPS14], [DPP18], ...

Protects search access patterns

Leaks search access patterns

Inefficient

Efficient
filesystem leakage is at the document level

End-to-end encrypted filesystem
FileSystem leakage is at the document level

End-to-end encrypted filesystem

Read doc 3
Filesystem leakage is at the document level

End-to-end encrypted filesystem

Read doc 3
Filesystem leakage is at the document level
Search access pattern leakage is at the word level
Search access pattern leakage is at the word level
Search access pattern leakage is at the word level

- End-to-end encrypted filesystem
- Leaky search system

Search "apple"
Search access pattern leakage is at the word level

End-to-end encrypted filesystem

Leaky search system

Search "apple"
Search access patterns can be used to recover document plaintext

File Injection Attack [ZKP16]

Enc(word₁) : Enc(doc₁), …
Enc(word₂) : Enc(doc₁₂), …
Enc(flu) :
  ...
Enc(wordₙ) : Enc(doc₅), …
Search access patterns can be used to recover document plaintext

File Injection Attack [ZKP16]

\[
\begin{align*}
\text{Enc(word}_1\text{)} & : \text{Enc(doc}_1\text{), ...} \\
\text{Enc(word}_2\text{)} & : \text{Enc(doc}_1\text{), ...} \\
\text{Enc(flu)} & : \\
\vdots & \text{Enc(word}_n\text{)} : \text{Enc(doc}_5\text{), ...}
\end{align*}
\]
Search access patterns can be used to recover document plaintext

Doc 27: “flu”

File Injection Attack [ZKP16]

\[
\begin{align*}
\text{Enc(word}_1): & \text{Enc(doc}_1), \ldots \\
\text{Enc(word}_2): & \text{Enc(doc}_12), \ldots \\
\text{Enc(flu)}: & \\
\vdots & \\
\text{Enc(word}_n): & \text{Enc(doc}_5), \ldots 
\end{align*}
\]
Search access patterns can be used to recover document plaintext

File Injection Attack [ZKP16]

Enc(word_1) : Enc(doc_1), …
Enc(word_2) : Enc(doc_{12}), …
Enc(flu) :
Enc(word_n) : Enc(doc_5), …
Search access patterns can be used to recover document plaintext

File Injection Attack [ZKP16]

Enc(word_1) : Enc(doc_1), ...
Enc(word_2) : Enc(doc_{12}), ...
Enc(flu) : Enc(doc_{27}), ...
::
Enc(word_n) : Enc(doc_5), ...
Search access patterns can be used to recover document plaintext.
Search access patterns can be used to recover document plaintext.

Repeat for all words in English dictionary.

File Injection Attack [ZKP16]

Enc(word_1) : Enc(doc_1), …
Enc(word_2) : Enc(doc_{12}), …
Enc(flu) : Enc(doc_{27}), …
::
Enc(word_n) : Enc(doc_{5}), …

“flu” = row 3

“flu” = row 3

Search access patterns can be used to recover document plaintext
Search access patterns can be used to recover document plaintext

File Injection Attack [ZKP16]

Enc(word₁) : Enc(doc₁), …
Enc(word₂) : Enc(doc₁₂), …
Enc(flu) : Enc(doc₂₇), …
⋮
Enc(wordₙ) : Enc(doc₅), …
Search access patterns can be used to recover document plaintext

File Injection Attack [ZKP16]

Enc(word_1) : Enc(doc_1), …
Enc(word_2) : Enc(doc_{12}), …
Enc(flu) : Enc(doc_{27}), …
…
Enc(word_n) : Enc(doc_{5}), …
Search access patterns can be used to recover document plaintext

File Injection Attack [ZKP16]

Enc(word₁) : Enc(doc₁), …
Enc(word₂) : Enc(doc₁₂), …
Enc(flu) : Enc(doc₂₇), …
⋮
Enc(wordₙ) : Enc(doc₅), …
Search access patterns can be used to recover document plaintext

File Injection Attack [ZKP16]

Enc(word_1) : Enc(doc_1), …
Enc(word_2) : Enc(doc_{12}), …
Enc(flu) : Enc(doc_{27}), Enc(doc_{82}), …
…
Enc(word_n) : Enc(doc_5), …
Search access patterns can be used to recover document plaintext

File Injection Attack [ZKP16]

Enc(word₁) : Enc(doc₁), …
Enc(word₂) : Enc(doc₁₂), …
Enc(flu) : Enc(doc₂₇), Enc(doc₈₂), …
…
Enc(wordₙ) : Enc(doc₅), …
Search access patterns can be used to recover document plaintext

... and many more attacks

[IKK12], [CGPR15], [KKNO16], [LZWT14], [PW16], [GTS17], [PWLP20], ....
Drawbacks of ORAM-based solutions

**ORAM**: client can read/write data at server and hide access patterns [GO96, SVSF+13].

Can implement search by building inverted index in ORAM.

+ Runtime logarithmic in index size.

- Large constants make cost prohibitive for encrypted filesystems.
DORY

ORAM-based solutions
[GO96], PathORAM, ....

Searchable Encryption (SE)
[SWP00], [Goh03], [CGKO11], [KPR12], [KP13], [CJJJ+14], [SPS14], [DPP18], ...

Protects search access patterns

Leaks search access patterns

Inefficient

Efficient
DORY

ORAM-based solutions
[GO96], PathORAM, …

Searchable Encryption (SE)
[SWP00], [Goh03], [CGKO11], [KPR12], [KP13], [CJJJ+14], [SPS14], [DPP18], …

Decentralized Oblivious Retrieval System

Protects search access patterns

Leaks search access patterns

Inefficient

Efficient
DORY eliminates search access pattern leakage

Search “apple”

End-to-end encrypted filesystem

Leaky search system
DORY eliminates search access pattern leakage

End-to-end encrypted filesystem

Search “apple”
To tackle this problem, we return to the system model:

**What do *real* encrypted filesystems require from a search system?**
Finding DORY: Identifying a system model 🐠

Surveyed 5 companies providing end-to-end encrypted filesystems.

Each wanted server-side search, but didn’t deploy because concerned about:

- Search access patterns
- Performance
Survey findings

See paper for full quantitative and qualitative findings.

- Requirements for latency, cost, and concurrency.
Survey findings

See paper for full quantitative and qualitative findings.

- Requirements for **latency**, **cost**, and **concurrency**.

Two most relevant findings:

1. **Linear scan for search is acceptable** if search latency and cost meet requirements for expected workloads.

2. **Distributing trust is acceptable** if certain security requirements are met.
Distributed trust

Provide security guarantees if an attacker can compromise some, but not all, trust domains.
Distributed trust requirements

At least one honest trust domain: attacker can't learn search access patterns.

• The other trust domains can be malicious.
Distributed trust requirements

At least one honest trust domain: attacker can't learn search access patterns.
  • The other trust domains can be malicious.

No honest trust domains: attacker can’t directly assemble search index.
  • Search access patterns are not protected.
Outline

1. DORY design

2. DORY evaluation
System architecture

[Simplified; does not account for replication]
Building DORY
### Search index [simplified]

<table>
<thead>
<tr>
<th>Doc 0</th>
<th>$x_{0,0}$</th>
<th>$x_{0,1}$</th>
<th>$x_{0,2}$</th>
<th>...</th>
<th>$x_{0,m}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc 1</td>
<td>$x_{1,0}$</td>
<td>$x_{1,1}$</td>
<td>$x_{1,2}$</td>
<td>...</td>
<td>$x_{1,m}$</td>
</tr>
<tr>
<td>Doc 2</td>
<td>$x_{2,0}$</td>
<td>$x_{2,1}$</td>
<td>$x_{2,2}$</td>
<td>...</td>
<td>$x_{2,m}$</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Doc $n$</td>
<td>$x_{n,0}$</td>
<td>$x_{n,1}$</td>
<td>$x_{n,2}$</td>
<td>...</td>
<td>$x_{n,m}$</td>
</tr>
</tbody>
</table>
**Search index** [simplified]

<table>
<thead>
<tr>
<th>Doc 0</th>
<th>$x_{0,0}$</th>
<th>$x_{0,1}$</th>
<th>$x_{0,2}$</th>
<th>$\cdots$</th>
<th>$x_{0,m}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc 1</td>
<td>$x_{1,0}$</td>
<td>$x_{1,1}$</td>
<td>$x_{1,2}$</td>
<td>$\cdots$</td>
<td>$x_{1,m}$</td>
</tr>
<tr>
<td>Doc 2</td>
<td>$x_{2,0}$</td>
<td>$x_{2,1}$</td>
<td>$x_{2,2}$</td>
<td>$\cdots$</td>
<td>$x_{2,m}$</td>
</tr>
<tr>
<td></td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>Doc $n$</td>
<td>$x_{n,0}$</td>
<td>$x_{n,1}$</td>
<td>$x_{n,2}$</td>
<td>$\cdots$</td>
<td>$x_{n,m}$</td>
</tr>
</tbody>
</table>

Bitmap for keywords in doc 1
Update [simplified]

update(docID, keywords)

- Client creates a bitmap for keywords.
- Client sends server the bitmap.
- Server updates the bitmap at row docID.

\[
\begin{array}{cccccc}
  x_{0,0} & x_{0,1} & x_{0,2} & \ldots & x_{0,m} \\
  x_{1,0} & x_{1,1} & x_{1,2} & \ldots & x_{1,m} \\
  x_{2,0} & x_{2,1} & x_{2,2} & \ldots & x_{2,m} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  x_{n,0} & x_{n,1} & x_{n,2} & \ldots & x_{n,m} \\
\end{array}
\]
Update [simplified]

update(docID, keywords)
• Client creates a bitmap for keywords.
• Client sends server the bitmap.
• Server updates the bitmap at row docID.
Update [simplified]

update(docID, keywords)

- Client creates a bitmap for keywords.
- Client sends server the bitmap.
- Server updates the bitmap at row docID.
Update [simplified]

update(docID, keywords)

- Client creates a bitmap for keywords.
- Client sends server the bitmap.
- Server updates the bitmap at row docID.
Search [simplified]

`search(keyword):`

- Client computes the index for `keyword` and sends to server.
- Server responds with corresponding column.
- Client outputs row numbers where column value is 1.

```
x_{0,0} x_{0,1} x_{0,2} \cdots x_{0,m}
x_{1,0} x_{1,1} x_{1,2} \cdots x_{1,m}
x_{2,0} x_{2,1} x_{2,2} \cdots x_{2,m}
\vdots \quad \vdots \quad \vdots \quad \vdots 
x_{n,0} x_{n,1} x_{n,2} \cdots x_{n,m}
```
Search [simplified]

search(keyword):

- Client computes the index for keyword and sends to server.
- Server responds with corresponding column.
- Client outputs row numbers where column value is 1.

\[
\begin{array}{cccc}
    x_{0,0} & x_{0,1} & x_{0,2} & \cdots & x_{0,m} \\
    x_{1,0} & x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\
    x_{2,0} & x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    x_{n,0} & x_{n,1} & x_{n,2} & \cdots & x_{n,m} \\
\end{array}
\]
Search [simplified]

search(keyword):
• Client computes the index for keyword and sends to server.
• Server responds with corresponding column.
• Client outputs row numbers where column value is 1.
Search [simplified]

search(keyword):

- Client computes the index for `keyword` and sends to server.
- Server responds with corresponding column.
- Client outputs row numbers where column value is 1.
**Search [simplified]**

`search(keyword):`

- Client computes the index for `keyword` and sends to server.
- Server responds with corresponding column.
- Client outputs row numbers where column value is 1.
Search [simplified]

search(keyword):
• Client computes the index for keyword and sends to server.
• Server responds with corresponding column.
• Client outputs row numbers where column value is 1.

GetIndex(keyword) → 2
Output matches
Challenge #1: Hiding search access patterns

Attacker learns search access patterns.

- Column requested leak data about keyword searched for.

\[
\text{getIndex}(\text{keyword}) \rightarrow 2 \\
\text{Output matches}
\]
Tool: Distributed Point Functions (DPFs) \cite{GI14, BGI15, BGI16}

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) [GI14, BGI15, BGI16]

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.

Retrieve $a_2$
Tool: Distributed Point Functions (DPFs) \([\text{GI14, BGI15, BGI16}]\)

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) [GI14, BGI15, BGI16]

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) [GI14, BGI15, BGI16]

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) [GI14, BGI15, BGI16]

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) [GI14, BGI15, BGI16]

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) [GI14, BGI15, BGI16]

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) \[\text{[GI14, BGI15, BGI16]}\]

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) [GI14, BGI15, BGI16]

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) \cite{GI14, BGI15, BGI16}

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) [GI14, BGI15, BGI16]

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Tool: Distributed Point Functions (DPFs) \cite{GI14, BGI15, BGI16}

- Uses multiple servers to hide which element the user is retrieving.
- If at least one server is honest, an attacker cannot learn the index requested.
- Requires a linear scan over the entire array.
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns

\[ \text{GetIndex(keyword)} \rightarrow 2 \]
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns

GetIndex(keyword) → 2

K₁

K₂
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns.

GetIndex(keyword) → 2

\[K_1\]

\[K_2\]
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns.
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns

GetIndex(keyword) → 2
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns.
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns.

GetIndex(keyword) → 2

K₁

K₂
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns

GetIndex(keyword) → 2
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns

GetIndex(keyword) → 2

$K_1$

$K_2$
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns
Leveraging DPFs to search

If at least one trust domain is honest, DORY hides search access patterns
Challenge #2: Compressing the search index

A bitmap for every word in the English dictionary is long!

- The linear scan for search takes a long time...

```
Doc 0    x_{0,0}  x_{0,1}  x_{0,2}  \cdots  x_{0,m}
Doc 1    \textcolor{green}{x_{1,0}  x_{1,1}  x_{1,2}  \cdots  x_{1,m}}
Doc 2    x_{2,0}  x_{2,1}  x_{2,2}  \cdots  x_{2,m}
          \vdots    \vdots    \vdots    \vdots
Doc n    x_{n,0}  x_{n,1}  x_{n,2}  \cdots  x_{n,m}
```
Using Bloom filters to compress the search index

**Bloom filters** provide efficient membership testing

<table>
<thead>
<tr>
<th>Apple</th>
<th>Orange</th>
</tr>
</thead>
</table>

\[
\begin{align*}
    x_{0,0} & \ x_{0,1} & \ x_{0,2} & \ldots & x_{0,m} \\
    x_{1,0} & \ x_{1,1} & \ x_{1,2} & \ldots & x_{1,m} \\
    x_{2,0} & \ x_{2,1} & \ x_{2,2} & \ldots & x_{2,m} \\
    \vdots & \ \vdots & \ \vdots & \ddots & \ \vdots \\
    x_{n,0} & \ x_{n,1} & \ x_{n,2} & \ldots & x_{n,m} \\
\end{align*}
\]
Using Bloom filters to compress the search index

**Bloom filters** provide efficient membership testing

```
Apple Orange
```

```
Apple 0 0 0 0 0 0 0 0 0 0
```

```
x_{0,0} x_{0,1} x_{0,2} \cdots x_{0,m}
x_{1,0} x_{1,1} x_{1,2} \cdots x_{1,m}
x_{2,0} x_{2,1} x_{2,2} \cdots x_{2,m}
\vdots \quad \vdots \quad \vdots \quad \vdots
x_{n,0} x_{n,1} x_{n,2} \cdots x_{n,m}
```
Using Bloom filters to compress the search index

**Bloom filters** provide efficient membership testing

```
Apple
Orange
```

```
x_{0,0} x_{0,1} x_{0,2} \ldots x_{0,m}
x_{1,0} x_{1,1} x_{1,2} \ldots x_{1,m}
x_{2,0} x_{2,1} x_{2,2} \ldots x_{2,m}
\vdots \quad \vdots \quad \vdots \quad \ddots
x_{n,0} x_{n,1} x_{n,2} \ldots x_{n,m}
```
Using Bloom filters to compress the search index

Bloom filters provide efficient membership testing

Apple
Orange

Apple

0 0 0 1 0 0 0 1 0 0

\[ x_{0,0} \ x_{0,1} \ x_{0,2} \ \cdots \ x_{0,m} \]
\[ x_{1,0} \ x_{1,1} \ x_{1,2} \ \cdots \ x_{1,m} \]
\[ x_{2,0} \ x_{2,1} \ x_{2,2} \ \cdots \ x_{2,m} \]
\[ \vdots \ \vdots \ \vdots \ \vdots \]
\[ x_{n,0} \ x_{n,1} \ x_{n,2} \ \cdots \ x_{n,m} \]
Using Bloom filters to compress the search index

**Bloom filters** provide efficient membership testing

<table>
<thead>
<tr>
<th>Apple</th>
<th>Orange</th>
<th>Orange</th>
<th>0 0 0 1 0 0 0 1 0 0</th>
</tr>
</thead>
</table>

\[
x_{0,0} \ x_{0,1} \ x_{0,2} \ \cdots \ x_{0,m} \\
x_{1,0} \ x_{1,1} \ x_{1,2} \ \cdots \ x_{1,m} \\
x_{2,0} \ x_{2,1} \ x_{2,2} \ \cdots \ x_{2,m} \\
\vdots \quad \vdots \quad \vdots \quad \quad \vdots \\
x_{n,0} \ x_{n,1} \ x_{n,2} \ \cdots \ x_{n,m}
\]
Using Bloom filters to compress the search index

**Bloom filters** provide efficient membership testing

- **Apple**
- **Orange**

![Diagram]

\[
\begin{align*}
x_{0,0} & \quad x_{0,1} \quad x_{0,2} \quad \cdots \quad x_{0,m} \\
x_{1,0} & \quad x_{1,1} \quad x_{1,2} \quad \cdots \quad x_{1,m} \\
x_{2,0} & \quad x_{2,1} \quad x_{2,2} \quad \cdots \quad x_{2,m} \\
\vdots & \quad \vdots \quad \vdots \quad \ddots \quad \vdots \\
x_{n,0} & \quad x_{n,1} \quad x_{n,2} \quad \cdots \quad x_{n,m}
\end{align*}
\]
Using Bloom filters to compress the search index

**Bloom filters** provide efficient membership testing

\[
\begin{align*}
\text{Apple} & \quad \text{Orange} \\
\text{Orange} & \quad 0 0 1 1 0 1 0 1 0 0 \\
\end{align*}
\]

\[
\begin{align*}
x_{0,0} & \quad x_{0,1} \quad x_{0,2} \quad \cdots \quad x_{0,m} \\
x_{1,0} & \quad x_{1,1} \quad x_{1,2} \quad \cdots \quad x_{1,m} \\
x_{2,0} & \quad x_{2,1} \quad x_{2,2} \quad \cdots \quad x_{2,m} \\
\vdots & \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
x_{n,0} & \quad x_{n,1} \quad x_{n,2} \quad \cdots \quad x_{n,m}
\end{align*}
\]
Using Bloom filters to compress the search index

Bloom filters provide efficient membership testing

Apple
Orange
Orange 0 0 1 1 0 1 0 1 0 0

\[ x_{0,0} \ x_{0,1} \ x_{0,2} \ \cdots \ x_{0,m} \]
\[ x_{1,0} \ x_{1,1} \ x_{1,2} \ \cdots \ x_{1,m} \]
\[ x_{2,0} \ x_{2,1} \ x_{2,2} \ \cdots \ x_{2,m} \]
\[ \vdots \ \vdots \ \vdots \ \vdots \ \vdots \]
\[ x_{n,0} \ x_{n,1} \ x_{n,2} \ \cdots \ x_{n,m} \]
Using Bloom filters to compress the search index

**Bloom filters** provide efficient membership testing

+ Preserves search column alignment
+ Compression
+ No fixed dictionary
Challenge #3: Encrypting the search index

Attacker should not immediately learn the search index contents.

**Strawman:** Encrypt every bit in Bloom filter.

- Search index size blows up by factor of $\lambda \approx 128$. 

\[
\begin{align*}
x_{0,0} & \ x_{0,1} \ x_{0,2} \ \ldots \ x_{0,m} \\
x_{1,0} & \ x_{1,1} \ x_{1,2} \ \ldots \ x_{1,m} \\
x_{2,0} & \ x_{2,1} \ x_{2,2} \ \ldots \ x_{2,m} \\
\vdots & \ \vdots \ \vdots \ \vdots \\
x_{n,0} & \ x_{n,1} \ x_{n,2} \ \ldots \ x_{n,m}
\end{align*}
\]
**Challenge #3: Encrypting the search index**

Attacker should not immediately learn the search index contents.

**Strawman:** Encrypt every bit in Bloom filter.

- Search index size blows up by factor of $\lambda \approx 128$.

<table>
<thead>
<tr>
<th>$x_{0,0}$</th>
<th>$x_{0,1}$</th>
<th>$x_{0,2}$</th>
<th>...</th>
<th>$x_{0,m}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{1,0}$</td>
<td>$x_{1,1}$</td>
<td>$x_{1,2}$</td>
<td>...</td>
<td>$x_{1,m}$</td>
</tr>
<tr>
<td>$x_{2,0}$</td>
<td>$x_{2,1}$</td>
<td>$x_{2,2}$</td>
<td>...</td>
<td>$x_{2,m}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>$x_{n,0}$</td>
<td>$x_{n,1}$</td>
<td>$x_{n,2}$</td>
<td>...</td>
<td>$x_{n,m}$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Enc}_k(x_{0,0}) & \quad \text{Enc}_k(x_{0,1}) & \quad \text{Enc}_k(x_{0,2}) & \quad \cdots & \quad \text{Enc}_k(x_{0,m}) \\
\text{Enc}_k(x_{1,0}) & \quad \text{Enc}_k(x_{1,1}) & \quad \text{Enc}_k(x_{1,2}) & \quad \cdots & \quad \text{Enc}_k(x_{1,m}) \\
\text{Enc}_k(x_{2,0}) & \quad \text{Enc}_k(x_{2,1}) & \quad \text{Enc}_k(x_{2,2}) & \quad \cdots & \quad \text{Enc}_k(x_{2,m}) \\
\vdots & \quad \vdots & \quad \vdots & \quad \cdots & \quad \vdots \\
\text{Enc}_k(x_{n,0}) & \quad \text{Enc}_k(x_{n,1}) & \quad \text{Enc}_k(x_{n,2}) & \quad \cdots & \quad \text{Enc}_k(x_{n,m})
\end{align*}
\]
Challenge #3: Encrypting the search index

**Solution:** generate a unique one-time pad using document version number.

\[
\begin{align*}
    x_{0,0} & \quad x_{0,1} & \quad x_{0,2} & \cdots & \quad x_{0,m} \\
    x_{1,0} & \quad x_{1,1} & \quad x_{1,2} & \cdots & \quad x_{1,m} \\
    x_{2,0} & \quad x_{2,1} & \quad x_{2,2} & \cdots & \quad x_{2,m} \\
    \vdots & \quad \vdots & \quad \vdots & \cdots & \quad \vdots \\
    x_{n,0} & \quad x_{n,1} & \quad x_{n,2} & \cdots & \quad x_{n,m} \\
\end{align*}
\]

\[
\oplus
\]

\[
\begin{align*}
    \text{PRF}_k(0 | | \text{version}_0) \quad & \quad y_{0,0} \quad y_{0,1} \quad y_{0,2} \cdots \quad y_{0,m} \\
    \text{PRF}_k(1 | | \text{version}_1) \quad & \quad y_{1,0} \quad y_{1,1} \quad y_{1,2} \cdots \quad y_{1,m} \\
    \text{PRF}_k(2 | | \text{version}_2) \quad & \quad y_{2,0} \quad y_{2,1} \quad y_{2,2} \cdots \quad y_{2,m} \\
    \vdots \quad & \quad \vdots \quad \vdots \cdots \quad \vdots \\
    \text{PRF}_k(n | | \text{version}_n) \quad & \quad y_{n,0} \quad y_{n,1} \quad y_{n,2} \cdots \quad y_{n,m}
\end{align*}
\]
Challenge #4: Malicious attackers

Need to defend against attackers that can influence server behavior.

**Strawman:** MAC every bit

- Search index (and search time) blows up by factor of $\lambda$.

\[
\begin{align*}
  x_{0,0} & \quad x_{0,1} & \quad x_{0,2} & \cdots & \quad x_{0,m} \\
  x_{1,0} & \quad x_{1,1} & \quad x_{1,2} & \cdots & \quad x_{1,m} \\
  \vdots & & & \ddots & \ddots \\
  x_{n,0} & \quad x_{n,1} & \quad x_{n,2} & \cdots & \quad x_{n,m} \\

t_{0,0} & \quad t_{0,1} & \quad t_{0,2} & \cdots & \quad t_{0,m} \\
  t_{1,0} & \quad t_{1,1} & \quad t_{1,2} & \cdots & \quad t_{1,m} \\
  \vdots & & & \ddots & \ddots \\
  t_{n,0} & \quad t_{n,1} & \quad t_{n,2} & \cdots & \quad t_{n,m}
\end{align*}
\]
Challenge #4: Malicious attackers

Need to defend against attackers that can influence server behavior.

Solution: use aggregate MACs to keep a single MAC per column.

\[
\begin{array}{cccccc}
  t_0 & t_1 & t_2 & \ldots & t_m \\
  x_{0,0} & x_{0,1} & x_{0,2} & \cdots & x_{0,m} \\
  x_{1,0} & x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  x_{n,0} & x_{n,1} & x_{n,2} & \cdots & x_{n,m} \\
\end{array}
\]
Other contributions (see paper)
Other contributions (see paper)

1. Efficient user revocation
Other contributions (see paper)

2. Extension to oblivious filesystems

1. Efficient user revocation
Other contributions (see paper)

1. Efficient user revocation
2. Extension to oblivious filesystems
3. Efficient replication leveraging DORY’s cryptographic properties
Outline

1. DORY design

2. DORY evaluation
Evaluation setup

https://github.com/ucbrise/dory

Evaluated performance using Enron email dataset.

Two baselines:

- Plaintext search: inverted index without encryption
- ORAM baseline: inverted index in PathORAM [SVSF+13] (see paper)
Eliminating frequency leakage.

Implementation.

constant as the number of documents increases due to the fact is significantly lower than that of DORY and stays relatively is apparent in Figure of time is spent performing the linear scan at the server. This MAC for every bit in the Bloom filter. Since than DORY because the client does not have to generate a hours. Note that semihonest DORY has a faster update time concern in DORY where updates are processed in less than Update latency determines (1) how long it takes for updates to be reflected in search results of document keywords. In contrast, DORY simply uploads a single encrypted Bloom filter. Update latency shows that the update latency of baseline beyond that number of documents.

Evaluation on Enron email dataset.

Table shows the breakdown in search contents of the documents themselves, the performance of techniques described above. To evaluate di blocks in each bucket).

For each search, we fetch the maximum number of blocks in a document, we fetch the maximum number of blocks a keyword maps to. Similarly for each keyword we update to known attacks. To handle state, the next client's access may leak access patterns (e.g. if it searched for the same word as the previous client). To handle client failures. If a client fails after issuing operations it reaches the target latency set by the companies (Table ). Parallelism allows us to reach the target latency set by the companies (Table ). Parallelism improves DORY's performance on Enron email dataset. We treat updates as adding an entire email to the index. Because the Enron email dataset only has emails, we do not measure the ORAM and plaintext search baselines.

Server state.

While DORY's per-

Search latency

Search latency (s)

ORAM

DORY

Plaintext search

Search latency

Search latency (s)

# Documents

# Documents

2^{10}

2^{15}

2^{20}

10^{-1}

10^{1}

10^{3}

10^{3}

10^{1}

10^{-1}

10^{3}

10^{1}

10^{-1}

10^{3}

10^{1}

10^{-1}

10^{3}

10^{1}

10^{-1}
Search latency

![Graph showing search latency versus number of documents. The graph compares ORAM, DORY, and plaintext search. ORAM achieves the lowest latency, followed by DORY, and plaintext search has the highest latency. The y-axis represents search latency in seconds, ranging from 10^{-1} to 10^3. The x-axis represents the number of documents, ranging from 2^{10} to 2^{20}. ORAM is marked as 185x faster than DORY at 2^{20} documents.]
Effect of parallelism on search latency

![Effect of parallelism on search latency](image)

**Search latency (s)**

- **DORY (p=1)**
- **DORY (p=2)**
- **DORY (p=4)**

**# Documents**

- 0
- 0.5M
- 1M

**Effect of parallelism on search latency**

While DORY's per-document search latency grows linearly with the number of documents (Figure 7.2), we observe significant improvements in search latency with increasing parallelism, as shown in the graph. This is particularly evident in the case of DORY (p=1), where the search latency remains relatively flat compared to lower parallelisms.

**Keywords**

- Search latency
- Parallelism
- Document count
- DORY

**Graphical Representation**

The graph illustrates the search latency in seconds for different numbers of documents, comparing DORY with varying levels of parallelism (p=1, p=2, p=4). The y-axis represents the search latency, while the x-axis shows the number of documents, ranging from 0 to 1M. The graph highlights how DORY's search latency remains stable even as the number of documents increases, due to improved parallelization.

**Key Observations**

- **Linear Growth**: For DORY (p=1), the search latency grows linearly with the number of documents.
- **Parallelism Benefits**: With increasing parallelism (p=2, p=4), the search latency decreases significantly, demonstrating the efficiency gains from parallel processing.

This demonstrates that DORY is particularly well-suited for environments with high search workloads, where parallelization can significantly reduce search latency.
Effect of parallelism on search latency

Parallelism improves search latency by roughly a factor of p (degree of parallelism).
Throughput

![Throughput Graph](image)

50% updates, 50% searches
Throughout

50% updates, 50% searches
Conclusion

• DORY is an efficient search system that hides search access patterns.

• By re-examining the system model, DORY reconciles the tension between efficiency and search access patterns.

• Search should not be a barrier to adoption of end-to-end encrypted systems.
Conclusion

• DORY is an efficient search system that hides search access patterns.

• By re-examining the system model, DORY reconciles the tension between efficiency and search access patterns.

• Search should not be a barrier to adoption of end-to-end encrypted systems.

Thanks! 🐠

Emma Dauterman
edauterman@berkeley.edu
https://github.com/ucbrise/dory
References


