Hakuin: Injecting Brain into Blind SQL Injection

Jakub Pružinec
Cybersecurity Researcher, Nanyang Technological University
Jakub Pružinec
Nanyang Technological University, Singapore
pruzinec.jakub@ntu.edu.sg
@offbyfour

Quynh Anh Nguyen
Nanyang Technological University, Singapore
aqnguyen@ntu.edu.sg
@capstone_engine
Agenda

1. SQL Injection & Blind SQL Injection
2. Optimizations & Tools
3. Language in Databases
4. Hakuin Framework
5. Performance Comparison
6. Future Work & Conclusion
Benign Interaction
SQL queries build from user input.

SQL Injection (SQLI)
Malicious input alters the query’s logic.

Blind SQL Injection (BSQLI)
Same, but content not exposed.
Response differences – yes/no questions.
Slow and suspicious – one bit per request
OPTIMIZATIONS

Exhaustive Search

Binary Search
Is the first letter in the range from “A” to “N”? Logarithmic complexity.

Character Set Narrowing
Try only certain characters (e.g., digits)

String Guessing
Try whole strings (e.g., common table names)

TOOLS

State of the Art
SQLMap, BBQSQL, jSQL Injection. Many features (e.g., vulnerability scanning). Rely on binary search for BSQLI (inefficient).
Natural Language
Text in DB is mostly in natural language.

Non-uniform character distribution
“A” is more common than “X” in English.

Context matters
The letter following “HELLO WORL_” is likely to be “D” but not “X”.

Binary Search not suitable
It treats all letters the same.
Hakuin
Framework for optimizing text extraction via BSQLI. Uses probabilistic language models & statistics.

Two approaches
One for DB schemas & one for DB content (i.e., rows)
Approach
A pretrained model estimates character probabilities based on partially extracted strings.
The probabilities are used to construct a Huffman tree.
The tree is searched – is the character in the left/right subtree?
Searching a well constructed Huffman tree is much faster than binary search.

Language Model
Five-gram trained on 2M tables and 3.8M columns extracted from Stack Exchange questions.

Detecting the End of String
EOS symbol predicted by the model and treated as any other character.
Much faster than extracting the string length in advance with binary search (other tools).
Approach
Two parts – string guessing & character extraction.
Problem 1: the data is not available in advance
We cannot pretrain models, so we train them on the fly.

Problem 2: Some models work well only on a certain type of data
We keep performance statistics of different strategies and always choose the best one. The statistics are available with no extra cost, because they are calculated once the correct character is already known.

Strategies
*Unigram* learns character distribution.
*Five-gram* learns patterns.
Binary Search is a fallback.
Strings in columns repeat
We keep track of previously extracted strings and try them again.

Approach
We construct a Huffman tree from the previous strings and search it.

Not all strings are worth trying
Adding a string to a Huffman tree raises the chances of success but increases the search cost.
We chose strings with high potential that minimize the expected number of requests (see the paper).

\[
\hat{c}_c = lr \\
\hat{E}(G) = P(x \in G)\hat{i}(G) + (1 - P(x \in G))\hat{c}_c \\
\hat{i}(G) = \sum_{g \in G} h_g p_g
\]
Measurements
Performance on DB schemas.
Performance on DB content.
Performance throughout the extraction process.

Datasets
*SchemaDB* dataset for RQ1 – 20 schemas, 184 tables, 938 columns, 12k characters.
*GenericDB* dataset for RQ2 and RQ3 – 4 tables, 12 columns, 1000 rows of real/realistic data.

Setup
A web application vulnerable to BSQLI.
Keeps count of the requests.

Tools
Hakuin, SQLMap, BBQSQL, jSQL Injection.
Performance on DB Schemas
Hakuin achieves 2.19 RPC, which is 5.98 times more efficient than the second-best tool.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Requests</th>
<th>RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hakuin</td>
<td>27123</td>
<td>2.19</td>
</tr>
<tr>
<td>SQLMap</td>
<td>167882</td>
<td>13.55</td>
</tr>
<tr>
<td>BBQSQL</td>
<td>162240</td>
<td>13.10</td>
</tr>
<tr>
<td>jSQL Injection</td>
<td>212225</td>
<td>17.13</td>
</tr>
</tbody>
</table>
Performance on DB Content
Compared to the second-best performing tool, Hakuin is up to 25.9 times more efficient on columns with limited values and up to 3.2 times faster on normal columns.

<table>
<thead>
<tr>
<th></th>
<th>users.first_name</th>
<th>users.last_name</th>
<th>users.sex</th>
<th>users.address</th>
<th>users.username</th>
<th>users.password</th>
<th>users.email</th>
<th>products.name</th>
<th>products.description</th>
<th>products.category</th>
<th>posts.text</th>
<th>comments.text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hakuin</td>
<td>4.88</td>
<td>5.33</td>
<td>0.32</td>
<td>2.19</td>
<td>5.75</td>
<td>4.28</td>
<td>3.74</td>
<td>3.87</td>
<td>3.22</td>
<td>0.43</td>
<td>4.3</td>
<td>3.91</td>
</tr>
<tr>
<td></td>
<td>(27899)</td>
<td>(32605)</td>
<td>(1602)</td>
<td>(86796)</td>
<td>(42185)</td>
<td>(137116)</td>
<td>(77910)</td>
<td>(490159)</td>
<td>(965824)</td>
<td>(6699)</td>
<td>(408165)</td>
<td>(345561)</td>
</tr>
<tr>
<td>SQLMap</td>
<td>8.19</td>
<td>8.11</td>
<td>8.30</td>
<td>6.92</td>
<td>7.98</td>
<td>7.39</td>
<td>7.15</td>
<td>6.75</td>
<td>6.42</td>
<td>7.45</td>
<td>6.7</td>
<td>6.58</td>
</tr>
<tr>
<td></td>
<td>(46820)</td>
<td>(49652)</td>
<td>(41502)</td>
<td>(274008)</td>
<td>(58569)</td>
<td>(236432)</td>
<td>(148871)</td>
<td>(856076)</td>
<td>(1923691)</td>
<td>(116585)</td>
<td>(636013)</td>
<td>(581155)</td>
</tr>
<tr>
<td>BBQSQL</td>
<td>13.15</td>
<td>12.44</td>
<td>12.91</td>
<td>10.4</td>
<td>13.33</td>
<td>10.23</td>
<td>10.3</td>
<td>10.25</td>
<td>-</td>
<td>10.74</td>
<td>-</td>
<td>7.48</td>
</tr>
<tr>
<td></td>
<td>(75154)</td>
<td>(76099)</td>
<td>(64550)</td>
<td>(411618)</td>
<td>(97862)</td>
<td>(327442)</td>
<td>(214388)</td>
<td>(1299591)</td>
<td>(28333)</td>
<td>(168036)</td>
<td>(972828)</td>
<td>(660765)</td>
</tr>
<tr>
<td>jSQL Injection</td>
<td>-</td>
<td>14.56</td>
<td>-</td>
<td>-</td>
<td>13.47</td>
<td>9.25</td>
<td>9.93</td>
<td>-</td>
<td>7.69</td>
<td>-</td>
<td>8.43</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(72402)</td>
<td>(89074)</td>
<td>(282)</td>
<td>(334874)</td>
<td>(98850)</td>
<td>(296122)</td>
<td>(206674)</td>
<td>(1008019)</td>
<td>(2302206)</td>
<td>(3666)</td>
<td>(799995)</td>
<td>(759075)</td>
</tr>
</tbody>
</table>
Performance Throughout Extraction Process

Hakuin’s models adapt quickly and outperform binary search almost immediately. In most cases, they performance continues to improve throughout the inference.
DEMO
Future Work
Near future – parallelism, pre-implemented DBMS queries (SQLite for now), non-textual data. Future – integration with SQLMap vs new tool?

Takeaways
New datasets (security lists)
- 300k unique tables, 700k unique column names, 6k DB names
- Available at https://github.com/pruzko/hakuin/tree/main/data/corpora

New language models
- Tables and columns pre-trained models
- Available at https://github.com/pruzko/hakuin/tree/main/data/models

New BSQLI framework Hakuin
- Available at https://github.com/pruzko/hakuin

Conclusion
BSQLI is slow but can be optimized. Language-aware and statistics-aware optimizations matter.
THANK YOU!