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

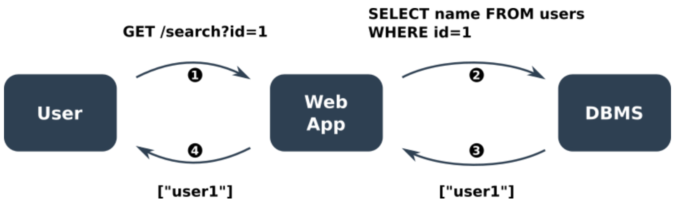
aqnguyen@ntu.edu.sg

@capstone_engine

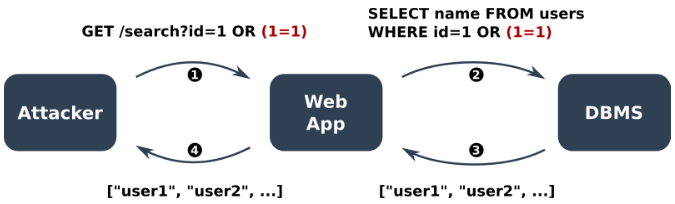
Agenda

1. SQL Injection & Blind SQL Injection
2. Optimizations & Tools
3. Language in Databases
4. Hakuin Framework
5. Performance Comparison
6. Future Work & Conclusion

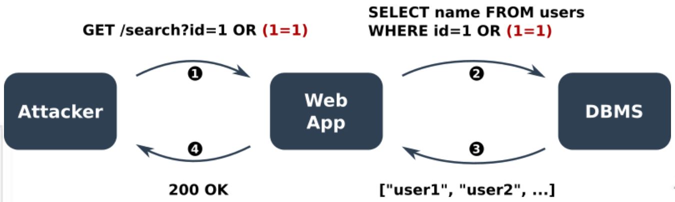
Benign Interaction
SQL queries build from user input.



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
Is the first letter “A”? Is it “B”? Is it “C”?
Linear complexity.

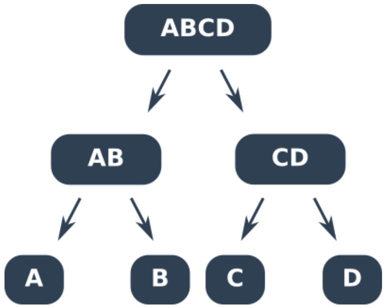
Binary Search
Is the first letter in the range from “A” to “N”?
Logarithmic complexity.

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.

	username	first_name	last_name	sex
	Filter	Filter	Filter	Filter
1	giamozz	Grady	Foley	male
2	machmudalo	Paula	Roberson	female
3	imtiyaz	Lynda	Gill	female
4	robmacliam	Andre	Ellison	male
5	andrew_sl	Caroline	Morales	female
6	ihtsl00	Ross	Travis	male
7	rom1	Angel	Valenzuela	male

Hakuin

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.

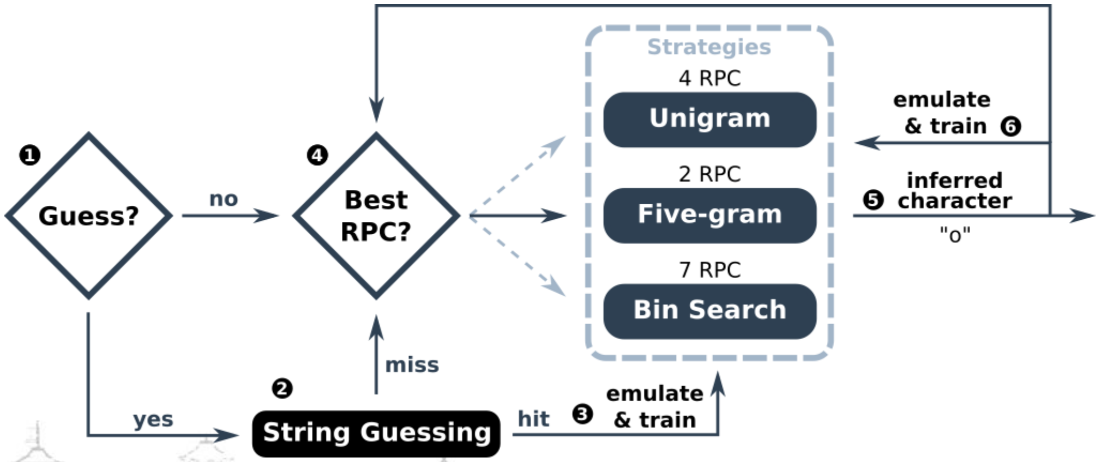
Detecting the End of String
EOS symbol predicted by the model and treated as any other character.
Much faster than extracting the string length in advance with binary search (other tools).

```
// select a character (refered to it as <x>)
Pseudo: x
SQL: substr(<column>, <idx>, 1)

// check if a character belongs to a list
Pseudo: x in ['a', 'b', 'c']
SQL: SELECT instr("abc", <x>) FROM ...

// check if a character belongs to a list with EOS
// <x> resolves to "" when <idx> exceeds length
Pseudo: x in [EOS, 'a', 'b']
SQL: SELECT <x> == "" OR instr("ab", <x>) FROM ..
```

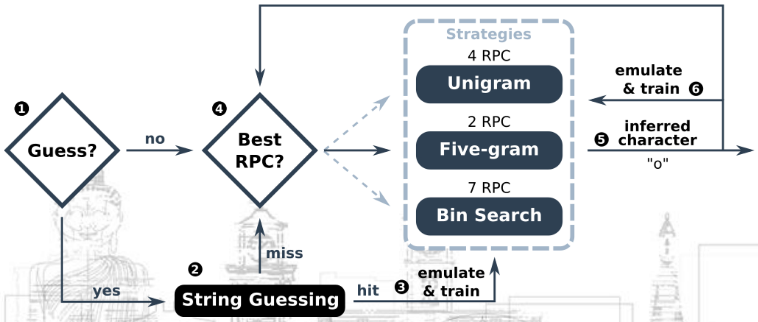
Approach
Two parts – *string guessing & character extraction*.



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.

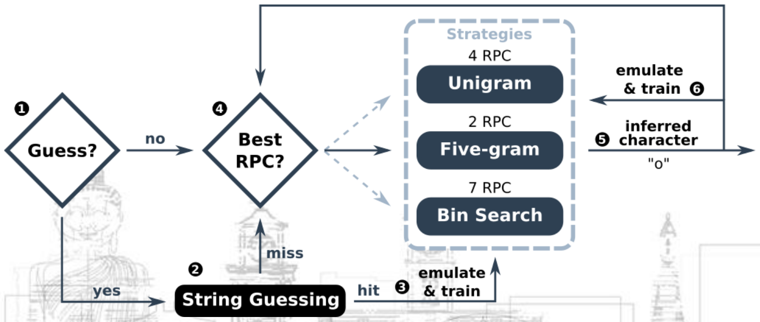


Strings in columns repeat
We keep track of previously extracted strings and try them again.

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).

```
// check if a string belongs to a list
Pseudo: <s> in ["guess1", "guess2"]
SQL: SELECT <s> in ("guess1", "guess2") FROM ...
```



$$\hat{e}_c = lr$$

$$\hat{E}(\mathbb{G}) = P(x \in \mathbb{G})\hat{t}(\mathbb{G}) + (1 - P(x \in \mathbb{G}))(\hat{t}(\mathbb{G}) + \hat{e}_c)$$

$$\hat{t}(\mathbb{G}) = \sum_{g \in \mathbb{G}} h_g p_g$$

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.

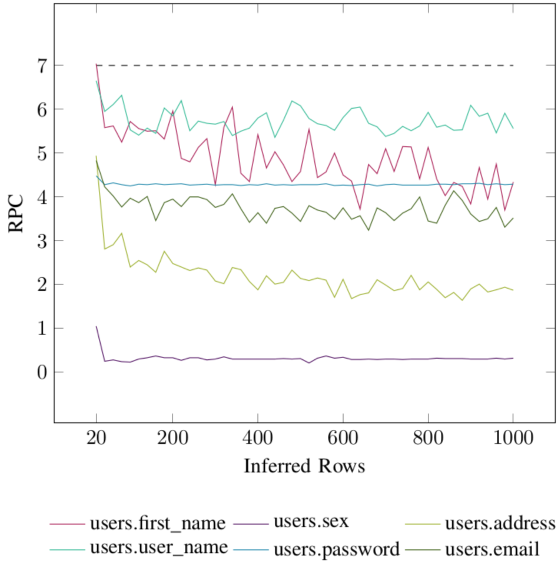
Tool	**Requests**	**RPC**
Hakuin	**27123**	**2.19**
SQLMap	167882	13.55
BBQSQL	162240	13.10
jSQL Injection	212225	17.13

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.

	users.first_name	users.last_name	users.sex	users.address	users.username	users.password	users.email	products.name	products.description	products.category	posts.text	comments.text
Hakuin	**4.88** (27899)	**5.33** (32605)	**0.32** (1602)	**2.19** (86796)	**5.75** (42185)	**4.28** (137116)	**3.74** (77910)	**3.87** (490159)	**3.22** (965824)	**0.43** (6699)	**4.3** (408165)	**3.91** (345561)
SQLMap	8.19 (46820)	8.11 (49652)	8.30 (41502)	6.92 (274008)	7.98 (58569)	7.39 (236432)	7.15 (148871)	6.75 (856076)	6.42 (1923691)	7.45 (116585)	6.7 (636013)	6.58 (581155)
BBQSQL	13.15 (75154)	12.44 (76099)	12.91 (64550)	10.4 (411618)	13.33 (97862)	10.23 (327442)	10.3 (214388)	10.25 (1299591)	- ~~(28333)~~	10.74 (168036)	- ~~(972828)~~	7.48 (660765)
jSQL Injection	- ~~(72402)~~	14.56 (89074)	- ~~(282)~~	- ~~(331874)~~	13.47 (98850)	9.25 (296122)	9.93 (206674)	- ~~(1008019)~~	7.69 (2302206)	- ~~(3666)~~	8.43 (799995)	- ~~(759075)~~

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 & MySQL 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/hakuin/data/corpora

New language models

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

New BSQLI framework Hakuin

- Available at https://github.com/pruzko/hakuin

Further Reading
Read our paper published at WOOT 23 - Hakuin: Optimizing Blind SQL Injection with Probabilistic Language Models

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

THANK YOU!