Hakuin: Optimizing Blind SQL Injection with Probabilistic Language Models

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

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

***Abstract*—SQL Injection (SQLI) is a pervasive web attack where a malicious input is used to dynamically build SQL queries in a way that tricks the database (DB) engine into performing unintended harmful operations. Among many potential exploitations, an attacker may opt to exfiltrate the application data. The exfiltration process is straightforward when the web application responds to injected queries with their results. In case the content is not exposed, the adversary can still deduce it using Blind SQLI (BSQLI), an inference technique based on response differences or time delays. Unfortunately, a common drawback of BSQLI is its low inference rate (one bit per request), which severely limits the volume of data that can be extracted this way.**

To address this limitation, the state-of-the-art BSQLI tools optimize the inference of textual data with binary search. However, this approach has two major limitations: it assumes a uniform distribution of characters and does not take into account the history of previously inferred characters. Consequently, the technique is inefficient for natural languages used ubiquitously in DBs.

This paper presents Hakuin - a new framework for optimizing BSQLI with probabilistic language models. Hakuin employs domain-specific pre-trained and adaptive models to predict the next characters based on the inference history and prioritizes characters with a higher probability of being the right ones. It also tracks statistical information to opportunistically guess strings as a whole instead of inferring the characters separately.

We benchmark Hakuin against 3 state-of-the-art BSQLI tools using 20 industry-standard DB schemas and a generic DB. The results show that Hakuin is about 6 times more efficient in inferring schemas, up to 3.2 times more efficient with generic data, and up to 26 times more efficient on columns with limited values compared to the second-best performing tool.

To the best of our knowledge, Hakuin is the first solution that combines domain-specific pre-trained and adaptive language models to optimize BSQLI. We release its full source code, datasets, and language models to facilitate further research.

I. Introduction

Modern web applications store data in databases (DB), usually relational ones, and retrieve it upon request. To fetch the data, a web application constructs an SQL query and sends it to the Database Management System (DBMS), a system responsible for managing the DB. The DBMS executes the query, retrieves the data, and responds. To ensure flexibility, web applications commonly allow user-supplied parameters and use them to build the queries dynamically. However, if done unsafely, i.e., the user input is directly inserted into an SQL query without validation, this can leave the application vulnerable to an attack called *SQL Injection* [1], [2], [3], [4] (SQLI). To exploit the vulnerability, an attacker crafts an input that, when embedded into the query, changes its logic. Consequently, they may be able to arbitrarily read and modify application data [5], [3], [1], [4], [2], bypass authentication [3], [1], access DBMS's file system [1], [2], abuse DBMS's networking capabilities [2], or even execute code remotely [3], [1], [2].

To prevent SQLI, OWASP's guidelines [6] recommend validating user input and using parameterized queries or stored procedures, where the user input is unambiguously recognized as a parameter rather than a part of the SQL statement. In addition, the guidelines advocate for abstraction layers and frameworks (e.g., object-relational mapping) that aim to assure safe and possibly automated query construction. Despite these efforts, SQLI vulnerabilities remain a prevalent issue in the contemporary web, being part of the third-ranking Injection category in OWASP's Top Ten in 2021 [7] as well as accounting for 1789 CVEs assigned in 2022 [8].

Once a vulnerability is discovered, attackers can extract data through various methods. Their first choice is to check if the web application responds to injected queries with its data and then read the data directly from the responses. If this is not an option, the attacker can inject invalid queries, induce DBMS error messages, and extract the data from them. However, this method only works if the web application forwards the error messages back to the attacker. Otherwise, the adversary can try to abuse DBMS's built-in networking capabilities to exfiltrate the data through an out-of-band channel. If nothing works, the attacker can still obtain the data with Blind SQLI (BSQLI) [3], [1], [2].

In BSQLI [9], [10], [11], [5], [12], [13], [14], the attacker injects a conditional query and infers its boolean value by observing response differences [3], [2], [14] (e.g., the HTTP status code) or measuring purposefully induced time-delays [5], [3], [4], [2] (e.g., with the `SLEEP` function). This allows the attacker to infer data by systematically testing all possible values until the right one is found. For instance, the attacker can determine the first character of a string by comparing it to all ASCII values and then repeat this process for each subsequent character until the whole string is obtained. However, this *exhaustive search* requires up to 127 requests per character (RPC), which is time-consuming and generates a lot of suspicious traffic. For example, to extract 1000 usernames that are 10 characters long, it needs 1.28 million requests. If a requests-response round-trip takes 500ms, the attack would

take about half a week on average, which is plenty of time for the defenders to detect and intervene with it. Even in case the attack goes unnoticed, the time requirements can be higher than what the attacker can afford. In other words, the efficiency of the search algorithm determines the amount of data that can be inferred with BSQLI.

To address this efficiency limitation, the state-of-the-art BSQLI tools [15], [16], [17] adopt *binary search* [11], [10], [13], [9], [14] as their main optimization technique. Instead of comparing each value individually, they determine if the right value lies within a specified range and gradually reduce this range until the value is found. Binary search significantly outperforms exhaustive search. However, when dealing with textual data, binary search treats all characters equally, which is not optimal for most text in DBs. This is because the text is written in natural languages, where characters occur with different frequencies (e.g., "a" is more common than "x" in English). Additionally, characters form strings where subsequent characters depend on preceding ones (e.g., "hell" is more likely to be followed by "o" than by "x"). Binary search ignores these factors, makes text inference unnecessarily slow, and consequently lowers the chance of the attack succeeding.

BSQLI optimization is largely overlooked with most research focusing on the prevention [18], [3], [19], [20] and detection [18], [3], [21], [22], [23], [24], [25], [26] of SQLI attacks as well as vulnerability testing [18], [3], [27], [28]. Roberto Salgado [9] suggests an optimization method that constructs a list of possible values and infers the index of the right one. Ruben Ventura [12] highlights the possibility of a vulnerable web application responding with multiple different responses, potentially leaking several bits of information at once. A similar approach described by Alex Chapman [10] combines response differences with time delays to infer multiple bits with a single request. These approaches focus on general data and completely ignore the lingual nature of the text in DBs. Pavel Sorokin [11] proposes using Huffman trees to prioritize frequently used characters, but this approach only considers characters in isolation, rather than in a sequence, and therefore does not model the language sufficiently. Lastly, Focardi *et al.* [14] optimize BSQLI with probabilistic binary search, but they rely on a generic bigram model trained on the Oxford dictionary, which only considers two-character sequences and does not cover the domain-specific language used in DBs, such as usernames.

Our research focuses on optimizing BSQLI inference of textual data with probabilistic language models that are trained on DB data and are therefore domain-specific. We utilize pre-trained and adaptive N-gram [29] models to predict the next characters and Huffman trees [30] to prioritize those that are likely to be the right ones. To train DB schema models, we collect SQL-related questions from Stack Exchange [31] and extract millions of table and column names. To model DB content, we use adaptive models and train them on the fly as we infer the data. Aside from inferring the data character by character, we use statistical information to opportunistically guess whole strings.

We implement our methods in a new framework called Hakuin. Hakuin abstracts away the optimization logic and allows users to easily automate BSQLI attacks. To assess its efficiency, we benchmark Hakuin against 3 state-of-the-art BSQLI tools [15], [16], [17] on 20 industry-standard DB schemas and a generic DB with 4 most common tables and 12 most common columns, each having 1000 rows filled with publicly available real data. The results show that Hakuin outperforms the second-best performing tool, being 5.98 times more efficient in inferring DB schemas, up to 3.2 times more efficient with generic data, and up to 25.9 times more efficient on columns with limited values.

In summary, we make the following contributions:

1) **Method** - to the best of our knowledge, we are the first to use domain-specific pre-trained and adaptive language models and opportunistic string guessing to optimize BSQLI.
2) **Implementation** - we implement our techniques in a framework called Hakuin. The framework is easy to use and allows users to effortlessly and efficiently automate BSQLI attacks. We benchmark Hakuin against 3 state-of-the art BSQLI tools in schema and content inference and show that it significantly outperforms all of the tools in both tasks. Hakuin is open-source and its codes are available to the public [32].
3) **Corpora and language models** - we extract millions of table and column names from Stack Exchange [31] questions and use them to train several language models. Again, the data is publicly available [32].

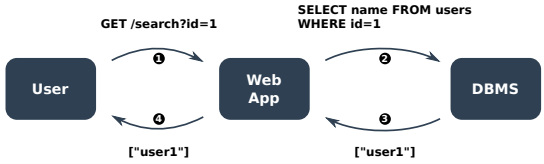
The rest of the paper is structured as follows. The background and related works can be found in Section II, the design of Hakuin in Section III, evaluation and results in Section IV, and finally, Section V concludes our work.

II. Background and Related Works

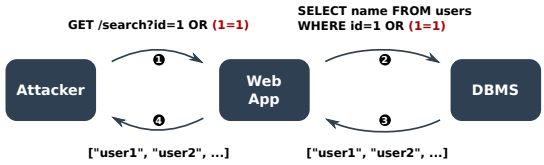
A. SQLI and BSQLI

The illustrative example in Figure 1 shows a vulnerable web application that implements a simple search functionality. There are three different scenarios depicted: a benign interaction, an SQLI attack, and a BSQLI attack. In the first benign scenario (Figure 1(a)), a user searches for someone by sending a request to the "`/search`" endpoint and providing the target user's id as a parameter (step ❶). The application uses this parameter to dynamically build an SQL query "`SELECT ... WHERE id=1`" (step ❷). The DBMS executes the query and returns the results (step ❸), which the application sends back to the user (step ❹).

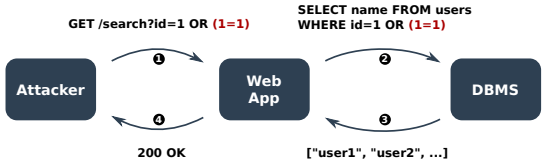
The problem arises when the application inserts the user-supplied parameter directly into the query without proper validation, which can lead to SQLI. In such case (Figure 1(b)), an attacker crafts a malicious payload "`1 OR (1=1)`" (step ❶), alters the query's logic (step ❷), and retrieves additional data (step ❸ and step ❹). Note that aside from the usernames, the adversary can obtain all data in all DB tables by injecting UNION queries (see [1], [2], [3] for the details).



(a) Benign



(b) SQLI



(c) BSQLI

Fig. 1. An illustrative example showing a benign interaction with a web application as well as SQLI and BSQLI attacks.

In the last BSQLI scenario (Figure 1(c)), the web application does not expose the search results, but rather indicates the search success with an appropriate HTTP status code (step ❹). The key difference here is that the attacker is able to inject queries but cannot read the results as they are not exposed. However, the HTTP response status codes are available to the attacker and reflect the boolean values of the injected queries. For example, because the adversary injects "`OR (1=1)`", which unconditionally resolves to logical "true", the application responds with "`200 OK`". In contrast, injecting "`OR (1=2)`" would resolve to "false", leading to a "`404 Not Found`" response. In other words, the attacker can inject arbitrary queries between the parentheses and deduce the boolean results from the response codes.

To infer textual data, the attacker can select a character and compare it to all possible values. In the example in Figure 1(c), the adversary can determine whether the first character of a username is "a" by injecting "`(SELECT substr(name,1,1) FROM users ...) = "a"`" in place of "`(1=1)`". They can then iteratively test other values until the character is found and repeat the process until the whole username is extracted. This approach, called *exhaustive search*, is inefficient as it requires up to 127 requests to infer a single ASCII character.

B. BSQLI Optimization Techniques

The current industry standard for BSQLI optimization is *binary search* [11], [10], [13], [9], [14]. This method begins with a range of possible values for a single character and divides it into two halves. A query is then injected to determine which half the right character belongs to. For example, if the query "`(SELECT substr(name,1,1) FROM users ...) < "n"`" resolves to "true", the character is in the lower half of the alphabet, otherwise it is in the upper half. The process repeats until the range is reduced to a single character, the right one. Unlike exhaustive search, binary search requires only $log_2(127) = 7$ requests to search the whole ASCII range. Another equivalent approach is to convert the character to its binary representation (e.g., with MySQL's `BIN` function) and use bitwise operations to infer individual bits [12], [10], [9]. This approach is, in essence, still a form of binary search, as the bitwise operations divide the set of possible characters into two halves based on their bit value at a given index. A downside of both methods is that they treat all characters equally, which is not efficient for natural language used ubiquitously in DBs. In contrast, our method uses language models to predict the next characters and Huffman trees to prioritize the most promising ones.

To optimize the search itself, BSQLI tools sometimes implement *character set narrowing* [11], [12], [10], [13], [9], a practice where the search range is reduced to a predefined set of characters. This can speed up the inference of columns that use only a subset of characters (e.g., password hashes consist of hexadecimal digits). However, the character sets need to be specified manually, which makes the technique hard to automate. In addition, the sets need to be known in advance, which is rarely the case. Our adaptive models, on the other hand, learn the character distributions from inferred data and therefore eliminate the need for user involvement.

Another possible optimization is to guess whole strings [14] or to infer only the first character and then guess the rest [10]. *String guessing* can be useful in obtaining frequently used table (e.g., "users") and column (e.g., "id") names as well as data in columns that have only a few possible values (e.g., "products.category"). The downside of this technique is that it introduces a miss penalty, making it hard to determine where the potential benefits outweigh the costs. Another disadvantage is that the guessable strings need to be known in advance. To overcome these limitations, we keep track of inferred strings and their statistics, which allows us to automatically identify where guessing is likely to succeed and requires no prior knowledge about which strings can be guessed.

Lastly, BSQLI is an I/O bound process where concurrency can massively speed up the inference [11], [12], [10], [14]. Consequently, all new optimization techniques should be designed with concurrency in mind. Our optimization method can be parallelized on a per-string basis.

C. N-gram Language Models

The goal of a character-based language model is to learn the probability distribution over strings, i.e., character sequences $w^n = (w_1, ..., w_n)$, where w_i denotes a character at position i. The probability of a string is defined as:

$$P(w^n) = P(w_1)P(w_2|w_1)...P(w_n|w^{n-1})$$

An *N-gram* [29] is a simplified language model that works with character sequences of length N. To estimate the proba-

bility of the following character, it only considers a history of $N-1$ last characters $h = (w_{n-1}, w_{n-2}, ..., w_{n-N})$ as:

$$P(w_n|w^{n-1}) \approx P(w_n|h)$$

These probabilities are learned during training, where the likelihood of observed sequences is maximized according to:

$$P(w_n|h) = \frac{c((h, w_n))}{\sum_w c((h, w))}$$

where c denotes the number of occurrences of a sequence. Simply put, N-grams estimate the probability of the next character based on the relative number of times it was observed in a given context.

The high-order N-grams sometimes fail to produce probabilities where their lower-order counterparts succeed. This is because some longer sequences are not observed during training but only their shorter parts are. For this reason, N-grams of all orders up to N are combined into a single *variable-length N-gram*[1]. To make a prediction, the sequence is gradually shortened and the order decreased until the model can be applied.

D. Related Works

Only a few studies relate to BSQLI optimization. Roberto Salgado [9] suggests a method called *bin2pos*. Bin2pos constructs a list of possible characters, determines the right character's position in the list, converts its index to a binary representation, and then infers it bit by bit. The key idea of this approach is that the index can be represented with fewer bits than the character itself. On the other hand, the length of the binary representations can vary and needs to be determined separately, which comes with a massive overhead. Ruben Ventura [12] introduces *lightspeed*, a technique that uses bitwise operations to infer several bits with a single request. This approach relies on web application endpoints that respond with more than two possible responses, essentially leaking multi-bit information. A similar approach described by Alex Chapman [10] takes advantage of combining response differences with purposefully induced time delays. While this multi-bit inference reduces the number of requests, it relies on delays that are unreliable and significantly increase the inference time. Pavel Sorokin [11] proposes using statistical information in combination with Huffman trees to prioritize frequently used characters. While this approach considers the character frequencies in a language, it does not take the history (i.e., previously inferred characters) into account. In addition, the author mentions this method only as an idea that has not been implemented or validated. Focardi *et al.* [14] optimize BSQLI with probabilistic binary search and a generic bigram language model. This approach can predict the next characters, however, it only considers a single-character history. Additionally, the model is trained on the Oxford dictionary, which only covers formal English but not the domain-specific DB language that includes data such as usernames. The authors [14] also propose word-based probabilistic binary search optimization, but this approach requires prior location of word boundaries and relies on a pre-specified word dictionary. In summary, the aforementioned solutions are either inefficient, work only under specific conditions, introduce request-time trade-offs, insufficiently model the language in DBs' textual data, or rely on prior knowledge. In contrast, our approach makes use of domain-specific language models to predict the next characters and Huffman trees to prioritize those that are more likely to be the right ones. Additionally, we opportunistically guess whole strings instead of inferring each character separately.

[1]A combined N-gram model is sometimes called an *everygram*.

Fig. 2. The process of inferring a single schema character.

There are several BSQLI tools available to the security community. *SQLMap* [15] is a versatile command line tool that infers lengths of strings and the strings themselves with binary search. It also tries to guess character sets of strings based on the first inferred character as well as to guess common table and column names. *BBSQL* [16] is a simple Python framework that utilizes binary search to infer strings and assumes the strings end when the search fails. Lastly, *jSQL Injection* [17] is a GUI-based tool that uses bitwise operations to infer strings and binary search to infer their lengths. These tools completely ignore the language used in the textual data.

III. Hakuin

Hakuin is a BSQLI optimization framework implemented in Python. It uses pre-trained models to infer DB schemas (i.e., table and column names) and adaptive models for DB content (i.e., rows of columns). The reason behind employing two different modeling approaches is that the schemas are similar across applications, while the content is mostly application-specific.

A. Schema Inference Overview

Figure 2 shows how a single schema character is inferred. First, a language model produces a likelihood distribution of possible next characters based on the partially inferred string, i.e., the history (step ❶). The probabilities are then used to construct a Huffman tree (step ❷). The tree is searched and a character is inferred (step ❸). The newly obtained character is then added to the history. This process is repeated until the whole string is extracted.

B. Schema Language Models

DB schemas usually share a lot of similarities across applications, e.g., the "users" table and the "id" columns are ubiquitous. To model a general schema, we collect table and column names[2] from Stack Exchange Data Dump [31] as follows:

[2]We also collect database, owner, and server names and publish them to promote further research. These corpora are not needed by Hakuin.

1) We select all posts from Stack Overflow that have at least one SQL-related tag ("sql", "db", or "database") and include all posts from Database Administrators.
2) We separate code from the posts, which is trivial because it is marked with "<code>" tags.
3) We split the code into lines, select those that contain at least one SQL keyword ("CREATE", "UPDATE", "INSERT", "CREATE", "ALTER", or "DROP"), and try to parse[3] them as separate SQL statements. The lines that are not successfully parsed are ignored.
4) We extract table and column names from the statements. If the *four part naming*[4] convention is used, we take out the tables and columns.
5) We remove duplicates on a per-question basis since they refer to the same table or column.
6) DBMS define a set of characters that can be used in valid object names and the others must be escaped. However, it is a common practice among programmers to escape even names that are valid and do not require it. Wherever possible, we unescape such names and discard the rest.
7) We convert all names to lowercase. This normalization is lossless as (most) DBMS are case-insensitive.
8) We clean up the data by removing invalid and non-ASCII names programmatically and dummy names manually. Because manual cleaning is tedious and time-consuming, we focus only on tables that occur more than 150 times and columns that occur more than 300 times.

This results in the creation of a corpus with 2M (295k unique) table names and a corpus with 3.8M (718k unique) column names. We use these corpora to train two variable-length five-grams (see Section II-C) and use them to model table and column names separately. Our preliminary experiments showed that increasing the order above 5 significantly increases the space complexity while providing no performance gain.

The models predict the following character or a special end of string (EOS) symbol. There are some nuances that need to be considered when constructing SQL queries with EOS (see Section III-G), but logically, EOS can be treated as any other character. Consequently, inferring the length of a string only requires extending the search space with one extra character. This is much more efficient than inferring the length with binary search in advance, which is what most of the other methods rely on.

C. Searching Huffman Trees

Hakuin constructs Huffman trees [30] based on model predictions and searches them to infer characters. The process starts at the root node and a list of all leaf nodes of the left subtree is made. A query (see Section III-G) is then injected to determine if the right character is in the list. If it is, the search moves to the left subtree, otherwise it moves to the right subtree. The process continues until a leaf node is reached - this node holds the right character. In case the character is not present in the tree, the algorithm stops at the rightmost leaf node. This situation arises when the character was not previously observed in the given sequence during training, and therefore the models do not predict it at all. Consequently, the rightmost leaf node is the only exception that requires an additional request to determine whether the search was successful. If not, the remaining characters not present in the tree are searched with binary search and the right character is inferred. Huffman trees prioritize the most promising leaf nodes as they have the shortest paths and thus require fewer requests to infer the characters they represent.

[3]SQL-metadata parser: https://pypi.org/project/sql-metadata/

[4]Four part naming: https://www.mssqltips.com/sqlservertip/1095/sql-server-four-part-naming/

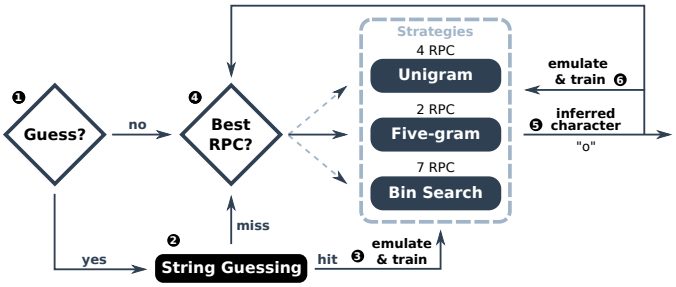


Fig. 3. The process of inferring a single content string.

It is possible to omit Huffman trees and search all characters gradually in order of their predicted likelihood, however, this is slower and turns into exhaustive search when the probabilities are roughly equal. Huffman trees, on the other hand, are balanced in this worst-case scenario and searching them results in a much faster binary search.

D. Content Inference Overview

Hakuin infers DB content on a per-column basis. To find the number of rows in a column, it starts with a default range $[0, 128)$ and injects a query (see Section III-G) to determine if the value falls within the range. If not, it gradually extends the range by a factor of two until it does. Lastly, it runs binary search over this range and infers the number of rows.

When inferring column rows (i.e., strings), Hakuin either guesses them in whole or it infers the strings character by character using several strategies. Figure 3 depicts how a single string is obtained. First, Hakuin decides whether to guess strings (step ❶). In case it does (step ❷) and succeeds, the correctly guessed string is used to emulate all strategies as well as to train their models (step ❸) and the process stops. Otherwise, Hakuin falls back to the per-character inference (starting at step ❹), where it selects the most efficient strategy and uses it to infer a single character (step ❺). Again, the character is used to emulate all strategies and train their models (step ❻). The per-character inference continues (at step ❹) until the EOS symbol is found and the whole string is extracted.

E. Strategies

As previously mentioned, Hakuin employs several strategies to infer content characters. The unigram and variable-

length five-gram strategies use their models the same way as described in the schema inference (see Section III-A and Figure 2). The only difference is that the models are not trained in advance but on the fly. Consequently, the more data Hakuin infers, the more accurate they get. The five-gram model recognizes patterns, while the unigram model is context-independent and only learns the character probability distribution, making it useful for dealing with random data that lack patterns. The last strategy, binary search, is used as a backup early on, when the models have not adapted yet.

The selection of strategies is based on their average RPC. For this reason, Hakuin keeps track of the statistics and updates them with every guessed string and inferred character. To compute the values, Hakuin emulates all strategies, meaning that it runs them as per usual, except without sending requests. This is possible, because at this stage of the process (see step ❸ and step ❻ in Figure 3), the right character is already inferred (or guessed) and all requests can be substituted with simple comparisons. This on-the-fly analysis allows Hakuin to automatically choose the most suitable strategy.

F. String Guessing

To guess strings, Hakuin keeps the history of all previously inferred strings and their counts, selects the most promising ones, constructs a Huffman tree based on the counts and searches it. Searching a string-based Huffman tree is conceptually the same as searching a character-based tree (see Section III-C), the only difference is that the injected queries work with strings rather than characters (see Section III-G).

Guessing strings makes sense only when the benefit (the requests saved with a successful guess) outweighs the cost (the requests necessary to search the tree). To decide when to try, Hakuin computes two values: $\hat{e}_c$, the expected number of requests necessary if guessing is skipped (resulting in per-character inference) and $\hat{E}_{min}$, the minimal expectation in case guessing takes place (resulting in guessing and possibly per-character inference). The first value is calculated as:

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

where l is the average length of previously inferred strings and r is the average RPC of the currently best strategy. The second value is determined through an iterative process where the previously inferred strings are gradually added to a candidate guessing set $\mathbb{G}$ (starting with the highest-count string) and its expectation $\hat{E}(\mathbb{G})$ is calculated. The process stops when the lowest expectation, $\hat{E}_{min} = \hat{E}(\mathbb{G}_{min})$, is found. The expectation is defined as:

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

where x is the right string and $\hat{t}$ is the expected number of requests necessary to search a Huffman tree. The probability $P(x \in \mathbb{G})$ is calculated based on the previously inferred strings. The expectation for searching a Huffman tree is:

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

where h_g is the path length to the node with string g and p_g is its probability calculated on previously inferred strings.

Hakuin tries guessing strings from $\mathbb{G}_{min}$ only when it is expected to be more efficient than per-character inference, i.e when $\hat{E}_{min} < \hat{e}_c$. However, columns with very long strings have high $\hat{e}_c$ which can make Hakuin guess strings even when it is likely to fail. Therefore, Hakuin requires $P(x \in \mathbb{G}) >= 0.5$ to prevent this issue.

G. Injected Queries

Listing 1 shows the types of queries Hakuin injects. A selection of a single character can be seen in Lines 2-3. Lines 6-7 determine whether the character belongs to a list of possible values. Hakuin uses these queries with binary search (see Section III-C and Section III-E) and to search Huffman trees (see Section III-C). Lines 11-12 show a variation with EOS included (see Section III-B). To guess strings, Hakuin constructs string-based Huffman trees (see Section III-F) and searches them with the queries in Lines 15-16. Lastly, Hakuin determines the number of rows in a column with numerical binary search (see Section III-D). Lines 19-20 are used to check whether the number of rows falls within a range.

All of these queries are simplified and do not handle problematic values, such as the non-printable characters. In Hakuin's implementation, we solve this by converting all characters and strings to their hexadecimal representations and treating them as raw blobs of data.

```
1  // select a character (refered to it as <x>)
2  Pseudo: x
3  SQL: substr(<column>, <idx>, 1)
4
5  // check if a character belongs to a list
6  Pseudo: x in ['a', 'b', 'c']
7  SQL: SELECT instr("abc", <x>) FROM ...
8
9  // check if a character belongs to a list with EOS
10 // <x> resolves to "" when <idx> exceeds length
11 Pseudo: x in [EOS, 'a', 'b']
12 SQL: SELECT <x> == "" OR instr("ab", <x>) FROM ..
13
14 // check if a string belongs to a list
15 Pseudo: <s> in ["guess1", "guess2"]
16 SQL: SELECT <s> in ("guess1", "guess2") FROM ...
17
18 // check if the number of rows is in a range
19 Pseudo: 0 <= len(<column>) < 128
20 SQL: SELECT count(*)>=0 AND count(*)<128 FROM ...
```

Listing 1. Examples of the SQLite queries Hakuin injects. The queries are written in a pseudo language for readability as well as in SQL.

IV. Evaluation

We aim to answer three research questions:

1) **RQ1**: How efficient is Hakuin in inferring DB schemas?
2) **RQ2**: How efficient is Hakuin in inferring DB content?
3) **RQ3**: How does Hakuin's performance change throughout the inference process?

To address these questions, we evaluate Hakuin and 3 other state-of-the-art BSQLI tools on two datasets - a schema dataset and a generic DB.

TABLE I
DESCRIPTIONS AND CHARACTER COUNTS OF THE GENERIC DB COLUMNS

Data	Description	Chars
users.first_name	The most common American first names. Half of them are male and half are female.	6k
users.last_name	The most common American last names.	6k
users.sex	User's sex. Half of the users are "male"and half are "female".	5k
users.address	Addresses of American start-up companies. An address consists of a street name and number, city, state, and a zip code.	40k
users.username	The most common usernames.	7k
users.password	MD5 hashes of the most common passwords.	32k
users.email	Email addresses generated using common combinations of users' first and last names. 45% of the addresses are provided by Gmail, 30% by Outlook, 10% by Yahoo, 10% by Apple, and 5% by other providers.	21k
products.name	Names of real products sold on Amazon.	127k
products.desc.	Descriptions of real products sold on Amazon.	300k
products.cat.	Categories of real products sold on Amazon. There are 20 categories and they are distributed unequally.	16k
posts.text	Real Twitter posts.	95k
comments.text	Real Reddit comments.	88k

A. Datasets

We construct the *schema dataset* from industry-standard DB schemas on Database Answers [33] as follows:

1) We crawl the web page and collect all images.
2) We manually select 20 entity relationship diagrams (ERD) and prioritize those with a larger number of entities and attributes.
3) We use Tesseract[5] to extract the names of entities and their attributes from the images and manually correct all mishandled characters.
4) We manually remove all unnecessary prefixes ("`<entity>_<attrib>`" is replaced with "`<attrib>`" and "`ref_<attrib>`" with "`<attrib>`").
5) We merge all entities of the same name into a single table. We remove duplicate attributes because all columns in a table must be unique.
6) We insert the tables into an SQLite DB.

In total, the schema dataset contains 184 tables, 938 columns, and 12388 characters.

Ideally, we would evaluate content inference on a real application DB, however the data is confidential and unavailable. As a way around, we construct a realistic *generic DB* from 4 most common tables and 12 most common columns in the corpora described in Section III-B. Afterwards, we fill every column with 1000 rows of publicly available data sourced from security lists, disclosed application DBs, and public services[6]. As of now, Hakuin only supports ASCII characters, therefore we limit the data to ASCII values only. Lastly, we convert the generic DB to SQLite format. Table I provides descriptions and character counts of the columns in the DB.

Both datasets are available in Hakuin's repository [32].

[5]Tesseract: https://pypi.org/project/pytesseract/

[6]The exact sources can be found in Hakuin's repository [32]

B. Experiment Setup

We set up a local web application with two endpoints that fetch data from the previously described SQLite DBs. Both endpoints accept a parameter vulnerable to BSQLI and return different HTTP status codes based on the query results. They also keep track of the number of requests sent to them, which allows us to objectively measure the number of requests necessary for inferring the DBs. Aside from Hakuin, we benchmark 3 state-of-the-art BSQLI tools: SQLMap version 1.6.11.7-dev [15], BBQSQL version 1.2 [16], and jSQL Injection version 0.84 [17]. Both SQLMap and jSQL Injection start the inference process with vulnerability scanning and DBMS fingerprinting, which generates additional requests. To filter these out, we reset the endpoint counters right before the actual inference process begins.

C. Results and Discussion

1) RQ1 - Schema Inference: Table II presents the results of the schema inference evaluation. As can be seen, Hakuin significantly outperforms the other tools in this task. Hakuin requires only 27123 requests to infer the schemas, whereas SQLMap, BBQSQL, and jSQL Injection require 167882, 162240, and 212225 requests, respectively. To infer a single schema character, Hakuin needs only 2.19 requests on average, while SQLMap needs 13.55, BBQSQL needs 13.10, and jSQL Injection needs 17.13. These numbers are surprisingly high considering they rely on binary search or bitwise operations, which require only 7 requests to search the whole ASCII range. The high number of requests needed by BBQSQL and jSQL Injection can be attributed to the inefficient implementation of their search algorithms, as they send unnecessary requests and repeatedly infer the same information. The reason behind SQLMap's high RPC is that it uses an alternative approach to infer DB schemas. DBMS often store SQL statements that were used to create DB objects in the DB itself. SQLMap infers these statements and extracts the table and column names from them. This is useful in situations where the attacker seeks to understand the DB constraints (e.g., the primary keys) or to read the code of the stored procedures they aim to exploit. However, inferring the table and column names directly is much more efficient and in most cases enough to dump the DB. Additionally, SQLMap and jSQL Injection infer the lengths of strings in advance and BBQSQL deduces the end of strings based on the search success. Both approaches are significantly less efficient than prediction EOS (see Section III-B), which is what Hakuin does.

Answer to RQ1: Hakuin infers DB schemas efficiently. To infer a single schema character, Hakuin requires only 2.19 requests on average, which is 5.98 times less than what the second-best performing tool needs.

2) RQ2 - Content Inference: Table III shows the results of the evaluation done on the generic DB. It presents the average RPC of the tools and the total number of requests required to infer each column. We observe that some tools occasionally produce incorrect data. Specifically, BBQSQL mishandles the

TABLE II
THE RESULTS OF DB SCHEMA INFERENCE EVALUATION

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

products.description and *posts.text* columns, where it stops the inference process prematurely and retrieves only partial strings. Meanwhile, jSQL Injection removes duplicates from columns before inferring them. This results in a loss of information, as the values are detached from their row indices and therefore cannot be associated with the corresponding entries in other columns. For instance, removing duplicates from *product.category* quickly reveals all the categories, but the information about which products belong to which category is lost. The columns jSQL Injection invalidates this way are *users.first_name*, *users.sex*, *users.address*, *products.name*, *products.category*, and *comments.text*. To avoid confusion, we cross out the request counts of all invalidly handled columns.

As can be seen, the efficiency of Hakuin heavily depends on the type of data it infers, ranging from 0.32 RPC to 5.75 RPC. The lowest RPC values, 0.32 and 0.43, are achieved on *users.sex* and *products.category* columns. This is because the columns have a limited number of possible values and Hakuin correctly identifies the potential to guess whole strings (see Section III-F) with a few requests. The performance on the other columns is largely determined by the patterns they exhibit. Columns *users.address* and *users.email* display high volumes of recurring patterns (e.g., the names of countries or mail providers) that Hakuin's five-gram model learns to recognize, resulting in 2.19 and 3.74 RPC respectively. Contrary to the other columns, *products.name*, *products.description*, *comments.text* and *posts.text* contain unstructured text but also exhibit patterns that the model can learn, resulting in 3.87, 3.22, 3.91, and 4.3 RPC respectively. On the other hand, columns *users.first_name*, *users.last_name*, and *users.user_name* exhibit few patterns, leading to 4.88, 5.33, and 5.75 RPC respectively. Lastly, the benefit of using multiple models and strategies is evident in the case of *users.password*. The MD5 hashes in the column appear random and have no patterns that the five-gram model can learn. However, the unigram model can at least learn the probabilistic distribution, i.e., the probability of hexadecimal digits being equal and the probability of other characters being zero. The model essentially identifies the effective character set which results in 4.28 RPC. Although a similar performance can be achieved using binary search and character set narrowing (see Section II-B), this approach is hard to automate as it relies on prior knowledge (the user must specify the character set) and only works well with uniformly distributed characters.

SQLMap's efficiency ranges from 6.42 to 8.30 RPC and roughly approaches the theoretical limit of binary search, i.e., 7 RPC. In contrast, the RPC of BBQSQL and jSQL Injection spans from 7.48 to 13.3 and 7.69 to 13.47 respectively, which significantly exceeds the expectation. Again, the poor performance of BBQSQL and jSQL Injection is a result of their inefficient implementation. Being the second-best performing tool, SQLMap achieves 8.30 and 7.45 RPC on *users.sex* and *products.category* respectively. In comparison, Hakuin requires 25.9 and 17.4 times less requests to guess the same data. The improvement over SQLMap on other columns varies depending on the patterns and distributions Hakuin can learn. The smallest and highest improvements are observed on *users.username* and *users.address*, where SQLMap achieves 7.98 and 6.92 RPC respectively. Hakuin needs 1.4 times less requests to infer the first column and 3.2 times less requests to infer the second one.

TABLE III
THE RESULTS OF DB CONTENT INFERENCE EVALUATION. THE TABLE SHOWS THE RPC AND THE TOTAL NUMBER OF REQUESTS NECESSARY TO INFER THE DIFFERENT COLUMNS OF THE GENERIC DB. INVALIDLY INFERRED COLUMNS ARE CROSSED OUT.

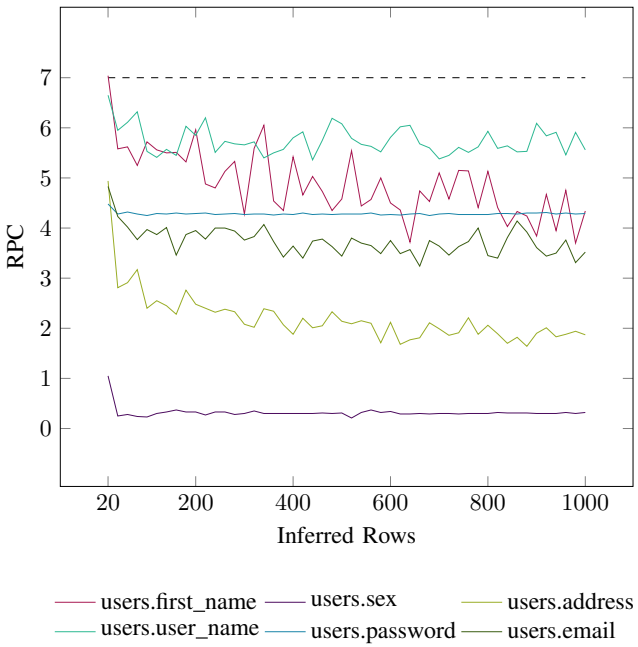
	users.first_name	**users.last_name**	**users.sex**	**users.address**	**users.username**	**users.password**	**users.email**	**products.name**	**products.description**	**products.category**	**posts.text**	**comments.text**
Hakuin	**4.88** (27899)	**5.33** (32605)	**0.32** (1602)	**2.19** (86796)	**5.75** (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)~~

Answer to RQ2: Hakuin's efficiency in inferring DB content varies based on the type of data it holds. Its average performance ranges from 0.32 to 0.43 RPC on columns with limited possible values, which is 17.4 to 25.9 times more efficient than the second-best performing tool. On generic columns, where values rarely repeat, Hakuin achieves an average RPC that spans from 2.19 to 5.75, and is 1.4 to 3.2 times more efficient than the second-best performing tool. Overall, Hakuin outperforms all other tools on all columns we tested.

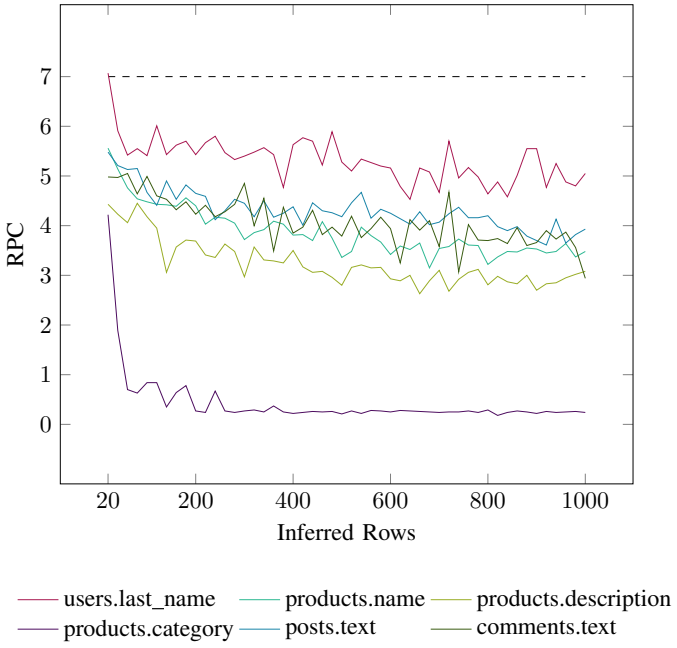
3) RQ3 - Evolving Performance: Lastly, we measure how Hakuin's performance evolves as its models adapt during the inference process. We compute the RPC on every column on a per-row basis, calculate their averages over 20 rows, and compare them with the 7 RPC baseline, i.e., the theoretical limit of binary search. We plot the results in Figure 4 and separate some representative columns (Figure 4(a)) from the rest (Figure 4(b)) to make the plots and the interpretation clearer. Note that due to the multi-row averaging, the graphs begin after the first 20 rows, i.e. once the models had some time to adapt. Before that, while they are still largely inaccurate, Hakuin emulates the model strategies, detects their low performance, and falls back to binary search (see Section III-E).

In most cases, the graphs start below the baseline and outperform binary search almost immediately within the first 20 rows. Being the only exception, the *users.first_name* column starts slightly above the baseline but outperforms binary search within the first 40 inferred rows as well. The *users.sex* column starts with a sharp drop from 1.05 RPC to 0.25 RPC within the first 40 rows and then remains relatively constant. This is because Hakuin is exposed to all possible strings in this column early on and starts successfully guessing them right away. The *users.password* column starts at 4.48 RPC, drops to 4.28 RPC within the first 40 rows, and stays nearly constant afterwards. This demonstrates that the unigram model needs less than 20 inferred MD5 hashes to accurately model the uniform distribution of the hexadecimal digits. The *users.first_name* and *users.address* columns show a substantial decline within the first 40 rows, followed by a slight downward trend. The *users.first_name* column starts at 7.07 RPC, drops to 5.58 RPC, and gradually decreases to approximately 4 RPC. Similarly, the *users.address* column drops from 4.94 RPC to 2.81 RPC and then slowly approaches 2 RPC. This is because the five-gram model adapts quickly in the early stage of the inference and continues to improve as it learns to recognize more patterns. The *users.username* and *users.email* columns also start with a steep decline within the first 40 rows, but the subsequent performance remains within a constant range. Specifically, the *users.username* column starts at 6.65 RPC and then stabilizes around 6 RPC, and the *users.email* column starts at 4.83 RPC and then stays around 3.5 RPC. This suggests that the model learns all available patterns at the beginning.

Answer to RQ3: Hakuin outperforms binary search on (almost) all columns within the first 20 inferred rows. When inferring columns with a limited number of possible strings, Hakuin reaches minimal RPC within the first 40 rows and maintains the performance afterwards. On generic columns, Hakuin's performance improves significantly within the first 40 rows and then, if the columns exhibit more patterns, continues to improve gradually throughout the whole process. In summary, Hakuin adapts in the early stage of the inference.



(a) The selected representative columns.



(b) The remaining columns.

Fig. 4. The evolution of Hakuin's performance as its models adapt.

V. Conclusion and Future Work

BSQLI is an SQLI variant where an attacker infers a DB based on response differences or purposefully induced time delays. The current state-of-the-art BSQLI tools use binary search as the primary optimization, however, this technique is not optimal for natural language in databases as it treats all characters equally and in isolation. To overcome this limitation, we developed Hakuin, a new BSQLI optimization framework that uses pre-trained and adaptive probabilistic language models to predict the next characters and Huffman trees to prioritize characters that are more likely to be the right ones. Aside from inferring the data character by character, Hakuin uses statistical information to opportunistically guess whole strings. We benchmarked Hakuin against 3 state-of-the-art BSQLI tools and showed that Hakuin is 5.98 times more efficient in inferring schemas, up to 3.2 times more efficient with generic data, and up to 25.9 times more efficient on columns with limited values compared to the second-best performing tool.

Hakuin currently targets only textual data but some techniques, such as statistical modeling and opportunistic guessing, can be applied to data of other types as well. Besides, while the content models are adaptive and language-independent, the schema models are trained on English data and are therefore limited to English. Future work should focus on optimizing the inference of non-English schemas and non-textual data. Lastly, our approach overlooks the semantics of column names even though it significantly affects the data they contain. We see a research direction that utilizes language processing techniques to understand column names and uses this information in combination with public data repositories to improve BSQLI even more.

Hakuin is open source and available to the public [32].

References

[1] C. Anley, "Advanced sql injection in sql server applications," NGSSoftware Insight Security Research (NISR), 2002.

[2] J. Clarke, *SQL injection attacks and defense*. Elsevier, 2009.

[3] W. G. Halfond, J. Viegas, A. Orso *et al.*, "A classification of sql-injection attacks and countermeasures," in *Proceedings of the IEEE international symposium on secure software engineering*, vol. 1. IEEE, 2006, pp. 13–15.

[4] C. Anley, "(more) advanced sql injection," NGSSoftware Insight Security Research (NISR), 2002.

[5] C. Alonso, D. Kachakil, R. Bordón, A. Guzmány, and M. Beltrán, "Time-based blind sql injection using heavy queries," 2017. [Online]. Available: https://media.defcon.org/DEF CON 16/DEF CON 2016 presentations/DEF CON 16 - alonso-parada-wp.pdf

[6] OWASP, "Sql injection prevention cheat sheet," 2021. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[7] ——, "Owasp: Top 10," 2021. [Online]. Available: https://owasp.org/Top10

[8] C. Details, "Security vulnerabilities published in 2022 (sql injection)," 2022. [Online]. Available: https://www.cvedetails.com/vulnerability-list/year-2022/opsqli-1/sql-injection.html

[9] R. Salgado, "Sql injection optimization and obfuscation techniques," 2013. [Online]. Available: https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2013/US-13-Salgado-SQLi-Optimization-and-Obfuscation-Techniques-WP.pdf

[10] A. Chapman, "Blind sql injection attacks optimization," 2017. [Online]. Available: https://ajxchapman.github.io/security/2017/01/14/blind-sql-injection.html

[11] P. Sorokin, "Fastest shot: Optimizing blind sql injection," 2022. [Online]. Available: https://hackmag.com/security/blind-sqli-optimize

[12] R. Ventura, "Blind sql injection attacks optimization," 2020. [Online]. Available: https://aircconline.com/csit/papers/vol10/csit101909.pdf

[13] R. Marcos, "Blind sql injection - optimization techniques," 2007. [Online]. Available: https://www.slideshare.net/amiable_indian/blind-sql-injection-optimization-techniques-214942

[14] R. Focardi, F. L. Luccio, and M. Squarcina, "Fast sql blind injections in high latency networks," in *2012 IEEE First AESS European Conference on Satellite Telecommunications (ESTEL)*. IEEE, 2012, pp. 1–6.

[15] B. Damele, M. Stampar, and D. Bellucci, "Sqlmap: Automatic sql injection and database takeover tool," 2006. [Online]. Available: https://sqlmap.org/

[16] CiscoCXSecurityLabs, "Bbqsql," 2012. [Online]. Available: https://github.com/CiscoCXSecurity/bbqsql

[17] ron190, "jsql injection," 2015. [Online]. Available: https://github.com/ron190/jsql-injection

[18] L. K. Shar and H. B. K. Tan, "Defeating sql injection," *Computer*, vol. 46, no. 3, pp. 69–77, 2012.

[19] S. W. Boyd and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks," in *Applied Cryptography and Network Security: Second International Conference, ACNS 2004, Yellow Mountain, China, June 8-11, 2004. Proceedings 2*. Springer, 2004, pp. 292–302.

[20] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove sql injection vulnerabilities," *Information and Software Technology*, vol. 51, no. 3, pp. 589–598, 2009.

[21] W. G. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter sql-injection attacks," in *Proceedings of the third international workshop on Dynamic analysis*, 2005, pp. 1–7.

[22] H. Gu, J. Zhang, T. Liu, M. Hu, J. Zhou, T. Wei, and M. Chen, "Diava: a traffic-based framework for detection of sql injection attacks and vulnerability analysis of leaked data," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 188–202, 2019.

[23] K. Wei, M. Muthuprasanna, and S. Kothari, "Preventing sql injection attacks in stored procedures," in *Australian Software Engineering Conference (ASWEC'06)*. IEEE, 2006, pp. 8–pp.

[24] D. Kar, S. Panigrahi, and S. Sundararajan, "Sqligot: Detecting sql injection attacks using graph of tokens and svm," *Computers & Security*, vol. 60, pp. 206–225, 2016.

[25] W. G. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter sql injection attacks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 175–185.

[26] G. Buehrer, B. W. Weide, and P. A. Sivilotti, "Using parse tree validation to prevent sql injection attacks," in *Proceedings of the 5th international workshop on Software engineering and middleware*, 2005, pp. 106–113.

[27] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: an input mutation approach," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 259–269.

[28] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *2009 IEEE 31st international conference on software engineering*. IEEE, 2009, pp. 199–209.

[29] P. F. Brown, V. J. Della Pietra, P. V. Desouza, J. C. Lai, and R. L. Mercer, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–480, 1992.

[30] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[31] S. Exchange, "Stack exchange data dump," 2021. [Online]. Available: https://archive.org/details/stackexchange

[32] J. Pružinec and Q. A. Nguyen, "Hakuin," 2023. [Online]. Available: https://github.com/pruzko/hakuin

[33] "Database answers," 2014. [Online]. Available: https://web.archive.org/web/20210508051645/http://www.databaseanswers.org/data_models/index_all_models.htm