A Query Reconstruction Attack on the Chase-Shen Substring-Searchable Symmetric Encryption Scheme

Zichen Gui
University of Georgia
zichen.gui@uga.edu

Kenneth G. Paterson
ETH Zurich
kenny.paterson@inf.ethz.ch

Sikhar Patranabis
IBM Research India
sikhar.patranabis@ibm.com

ABSTRACT

Searchable symmetric encryption (SSE) enables queries over symmetrically encrypted databases. To achieve practical efficiency, SSE schemes incur a certain amount of leakage; however, this leads to the possibility of *leakage cryptanalysis*, i.e., cryptanalytic attacks that exploit the leakage from the target SSE scheme to subvert its data and query privacy guarantees. Leakage cryptanalysis has been widely studied in the context of SSE schemes supporting either keyword queries or range queries, often with devastating consequences. However, little or no attention has been paid to cryptanalysing *substring-SSE* schemes, i.e., SSE schemes supporting arbitrary substring queries over encrypted data. This is despite their relevance to many real-world applications, e.g., in the context of securely querying outsourced genomic databases. In particular, the first ever substring-SSE scheme, proposed nearly a decade ago by Chase and Shen (PoPETS '15), has not been cryptanalysed to date.

In this paper, we present the *first* leakage cryptanalysis of the substring-SSE scheme of Chase and Shen. We propose a novel inference-based *query reconstruction attack* that: (i) exploits a *reduced* version of the actual leakage profile of their scheme, and (ii) assumes a *weaker* attack model as compared to the one in which Chase and Shen originally claimed security. We implement our attack and experimentally validate its success rate and efficiency over two real-world datasets. Our attack achieves high query reconstruction rate with practical efficiency, and scales smoothly to large datasets containing 100,000 strings.

To the best of our knowledge, ours is the first and only query reconstruction attack on (and the first systematic leakage cryptanalysis of) *any* substring-SSE scheme till date.

1 INTRODUCTION

Searchable Symmetric Encryption. Searchable symmetric encryption (SSE) [10, 13, 15, 26, 60] is a widely studied cryptographic primitive that supports efficient queries over symmetrically encrypted databases. SSE is a key enabler for secure storage-as-a-service, wherein clients can securely outsource the storage and processing of large databases to (potentially untrusted) third party servers. The goal of SSE is to enable efficient query processing *directly* over the encrypted database, while ensuring client privacy by minimizing information "leakage" to the server.

The vast majority of SSE schemes in the literature support *single keyword* queries (i.e., given an encrypted document collection in which each document is tagged with keywords, it is possible to find the set of all documents associated with a target keyword). Examples include [6, 7, 11–13, 15, 26, 43, 60]. SSE schemes supporting a richer class of Boolean queries over collections of keywords have also been studied [9, 10, 40, 50, 56, 58]. Other works have investigated SSE schemes for range queries [17–20], as well as for join and group-by queries over encrypted relational databases [17, 39, 41].

Substring-SSE. In this paper, we focus on applications where the client needs to query a database for *arbitrary substrings*, as opposed to predetermined keywords. Concretely, substring-searchable symmetric encryption (substring-SSE) aims to support the following query functionality: given an encrypted string (where the string is a sequence of characters picked from some fixed alphabet), return the positions of all occurrences of a target substring within this string.

As a motivating use-case, suppose that a medical research lab wishes to store the genomic data of subjects on a cloud server, while allowing researchers to issue substring queries to detect the existence of particular DNA sequences (e.g., to trace the existence of cancer marker sequences, or to determine the rarity of a potentially useful probe sequence). For such an application, it is essential to ensure the privacy of the database as well as the queries. An SSE scheme that supports arbitrary substring queries would offer a solution in this case.

The naïve approach of re-purposing SSE for single keyword search to construct substring-SSE (where each substring of a given string is modeled as a separate keyword) is inefficient in practice: for a length n string, the number of keywords required would be $O(n^2)$. Achieving practically efficient substring-SSE is much more challenging. Indeed, since the seminal work in this direction by Chase and Shen in PoPETS'15 [14], only a handful of SSE schemes for substring search have been proposed [20, 36, 51]. Of these, the solutions proposed in [14] and [51] are based on specialized encrypted data structures, while [20] and [36] model substring queries as special cases of conjunctive keyword queries and range queries, respectively, and obtain SSE schemes for substring search by adapting SSE schemes for those other types of query.

Leakage in SSE. The term "leakage" is used in the SSE literature to denote any information that the server learns about either the database itself or the client's queries. While optimally private SSE (with little or no leakage) can be achieved using fully homomorphic encryption (FHE) [24] and Oblivious RAM (ORAM) [27, 28], these techniques incur significant computational and/or communication overheads today, making them impractical at meaningful scale. Existing SSE designs opt for enhanced performance at the cost of leaking some information to the server [10, 13, 15]. A typical security proof for such a scheme establishes, via a simulation-based

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
© 2024 Copyright held by the owner/author(s).

security model, that the scheme in operation leaks no more than a certain well-defined leakage pattern.

This approach leaves open the question of whether the leakage pattern of a given scheme is acceptable or not in practice. This calls for an assessment of the real-world impact of that leakage, via *leakage cryptanalysis*. This involves developing concrete cryptanalytic attacks that exploit the leakage of the SSE scheme to subvert some security guarantee (such as data or query privacy). Such attacks are also known as *leakage abuse attacks*. Starting with the foundational work of Islam et al. [37], leakage cryptanalysis has been studied extensively in the context of SSE for keyword queries [2, 8, 16, 33, 34, 53–55, 59] and SSE for range queries [21, 30–32, 44–49, 52]. These attacks typically exploit one or more forms of leakage that commonly occur in SSE schemes, such as : (i) *access pattern leakage* (revealing the set of results matching a given query), (ii) *volume leakage* (revealing the number of results matching a given query), (iii) *co-occurrence pattern leakage* (revealing the number of common results across a pair of queries), and (iv) *search pattern leakage* (revealing whether two queries are identical). Today, leakage cryptanalysis forms an essential part of the overall security evaluation of SSE schemes.

Leakage in Substring-SSE. To the best of our knowledge, no prior work has carried out any leakage cryptanalysis of substring-SSE schemes. In particular, the seminal paper of Chase and Shen [14] lacks such an analysis. We note that attempting to cryptanalyze the leakage from substring queries throws up new challenges that do not arise in the context of SSE for keyword or range queries. For example, consider the query equality leakage from keyword queries. This leakage is essentially "all-or-nothing" – either two keywords are identical or they are not. This is particularly useful in many existing attacks that rely on query equality to (informally speaking) *filter* the potential set of keywords matching a given query [2, 8, 34]. However, in the case of substring queries, two substrings may have varying degrees of overlap in terms of the number of characters they have in common, which leads to the possibility of more nuanced, fine-grained query equality leakage, requiring advanced filtering approaches.

Moreover, unlike in the case of keyword queries, where the set of potential keywords is fixed *a priori* (and is typically in the range of millions for even large real-world document collections), arbitrary substring queries come from a significantly larger universe (for example, the number of possible strings of length 5 containing characters from the English alphabet is more than a billion). This limits the usefulness of previous analysis techniques for SSE schemes.

Given these novel challenges, the lack of prior analysis, and the central role of such analysis in assessing the real-world security of SSE schemes, we believe that leakage cryptanalysis of substring-SSE schemes is a problem deserving of attention.

1.1 Our Contributions

We perform the first leakage cryptanalysis of the seminal substring-SSE scheme due to Chase and Shen from PoPETS'15 [14] (henceforth, the CS scheme). More specifically, we propose a novel query reconstruction attack against the CS scheme. Our attack only makes use of a *reduced* version of the full leakage profile of the CS scheme (as formally established in [14]), and works in a *weaker* attack model as compared to the one in which Chase and Shen originally claimed their scheme to be secure. We implement our attack and experimentally validate its success rate and practical efficiency over two real-world datasets – English Wikipedia[1] and a genome dataset[2] from the National Center for Biotechnology Information (NCBI).

To the best of our knowledge, ours is the first query recovery attack on (and the first systematic leakage cryptanalysis of) any substring-SSE scheme. We focus on the CS scheme because it was the first substring-SSE scheme to be proposed, and notably, has not been the subject of leakage cryptanalysis over nearly a decade. It would be interesting to cryptanalyse the leakage of other substring-SSE schemes [20, 36, 51]; however, we believe that each one would likely require the development of its own, distinct cryptanalytic techniques, since they each rely on fundamentally different design principles and vary widely in their leakage profiles. We leave (crypt)analyzing these schemes as an interesting open question.

1.2 Technical Overview

We present a detailed technical overview of our contributions below.

Informal Overview of the CS Scheme. We begin with a very high-level and informal description of the CS scheme, with just enough detail for a meaningful exposition of our attack approach. We refer the reader to Section 3 for a more detailed treatment.

As mentioned earlier, the CS scheme is based on *suffix trees*. A suffix tree is a data structure that is popularly used to efficiently perform substring search on non-encrypted data. In a suffix tree-based representation of a string s, the path from the root node to every other node consists of a sequence of labeled edges that represents a unique suffix of s. Substring search is based on the following key observation: a string q is a substring of s if and only if q is a prefix of some suffix of s. Thus, searching for an occurrence of q in s reduces to identifying a path from the root to some *matching* node for q (i.e., a node such that the sequence of labels along the path from the root to this node matches q).

Building upon the above idea, the CS scheme uses a combination of basic symmetric-key cryptographic primitives to enable an *encrypted* substring search procedure that, given an *encrypted* substring query, allows traversal of select edges in an encrypted representation of the suffix tree. At a high level, the CS scheme creates this encrypted representation using a key-value dictionary. More concretely, each node u in the suffix tree is associated with a pseudorandom key representing the path from the root to u, and the associated value encrypts the index for u. The encrypted version of substring search now proceeds as follows: (i) compute the pseudorandom key for the node matching the query substring q, (ii) search the encrypted key-value store for the matching key, and (iii) if a match is found, retrieve and decrypt the corresponding index. We reiterate that this is a highly simplified description of the CS scheme and we are intentionally glossing over many technical details and optimizations, but should be sufficient to obtain a high-level overview of our attack approach.

Leakage from the CS Scheme. Our attack relies on three subcomponents of the overall leakage profile of the CS scheme, as

[1] https://dumps.wikimedia.org/simplewiki/
[2] https://www.ncbi.nlm.nih.gov/

established in [14]. We present a simplified and informal summary of these leakage sub-components (see Section 4 for details): (i) *path length*: the length of the path from the root to the matching node for any queried substring q (if such a matching node exists), (ii) *response volume*: the number of matching occurrences of any queried substring q, and (iii) *common prefix length*: the length of the *longest common prefix* for any pair of queried substrings (q, q').

We highlight that the overall leakage profile for CS is *significantly larger than* the above sub-components in combination. In fact, as we discuss in Section 4, a (passive) adversary can derive the above leakage components directly from just the transcript of messages exchanged by the client and the server during an execution of the search protocol of CS. In other words, the attack can be executed not only by an adversarial server, but *any* adversarial entity that can observe the communications between the client and the server.

Overview of Our Attack. We exploit the above leakage sub-components to design a query recovery attack against the CS scheme. The core principle of our attack is as follows: we show how to develop novel statistical models for substring queries based on the above leakage sub-components from CS and auxiliary data in the form of an approximate version of the original database. We then use the resulting models to develop a new inference-style leakage cryptanalysis-based attack that targets query reconstruction. We additionally highlight that our attack assumes an "honest-but-curious" (passively eavesdropping) server. While this is the most commonly assumed adversarial model in the broader SSE literature, the authors of [14] claimed security of CS in the much stronger "fully malicious server" model. Thus our attack works in a *weaker* attack model as compared to the one in which the CS scheme was originally claimed to be secure. We refer the reader to Section 4 for a more detailed description of the attack.

Auxiliary Data. Our attack makes use of auxiliary data, that is, additional data that is assumed to be distributed in the same way as that in target database. More precisely, we assume that the target database and the auxiliary database are sampled independently from the same distribution. This constitutes a weaker attack assumption as compared to "known-data attacks" (e.g. [2, 8, 34, 37]), where the auxiliary and target databases are assumed to be identical. The precise sampling strategy used in our experimental evaluation can be summarized as follows: given a dataset, we set aside half of it for use as the attack target, i.e. for generating the leakage, while using sub-samples of the other half to define auxiliary data. In this way, we effectively use independent samples from the same empirical distribution to define the target and auxiliary data distributions. This is akin to the standard approach of separating training data from test data that is used, for example, in machine learning. The approach is described in more detail in Section 5.

It could be argued that our approach of splitting the available data into two sets, one to create the auxiliary distribution, the other to use in experiments, is too idealised, in that we effectively give the adversary a known plaintext distribution. While this is true, we argue that a) the scheme should still hide queries even in this setting, b) our assumption is weaker than the known plaintext assumption used in related work [2, 8, 34, 37], and c) it could be relaxed by using two different (but related) distributions, or by adding noise to the auxiliary distribution. We leave the exploration of the latter options to future work.

Attack Idea. The core idea of our attack is to solve an optimization problem, where the objective function is the formal *likelihood* of observing a given assignment of candidate substrings to queries, given the observed leakage and auxiliary data as prior information. We then maximize the objective function using simulated annealing; this corresponds to maximizing the likelihood of the solution. Thus the simulated annealing, if it works, will produce "good" solutions in which many candidate substrings in the solution are correctly assigned to the corresponding queries. Of course, one could also use any performant optimization technique in place of simulated annealing. This approach requires careful mathematical analysis to derive the likelihood function. We build upon prior work on attacking SSE schemes for keyword search using similar approaches (such as [33]); the core technical foundation of our work is in adapting such analyses to the leakage from the CS scheme.

Single vs Multiple Strings. The original CS scheme only handles single strings, and we generalised it to handle multiple strings. Some of our experimental attacks were mounted in the multi-string setting. However, we wish to emphasise that our attacks work equally well in the single- and multi-string settings. In particular, their performance in the multi-string setting for a collection of strings whose *sum* of lengths is equal to L would be identical to that in the single-string setting for a string of length L. This is because in the multi-string setting, the scheme first builds one long string by concatenating all the strings from the collection using a special symbol to mark the end of each string; however, our attack never involves substrings containing this symbol.

Attack Implementation. We then show how to efficiently evaluate this likelihood function on large sets of queries and leakage. This is particularly challenging since, unlike prior inference-style attack approaches for recovering keyword queries (such as [16, 33, 54, 55]), the candidate sets for arbitrary substrings turn out to be *orders of magnitude larger*, even for medium-sized databases. To tackle this, we designed and implemented several computational optimization and approximation techniques that make our attack highly scalable with respect to the number of queried substrings, number of candidate substrings in the auxiliary information, and the size of our target dataset. We developed our own implementation of simulated annealing for speed and flexibility.

Experimental Evaluation. In Section 5, we present extensive experimental evaluations to validate the practicality of our proposed attack over English Wikipedia and the NCBI genome dataset. Our experiments show that our attack achieves high success rate with reasonable practical efficiency and while scaling to large datasets. For English Wikipedia ($100,000$ Wikipedia pages, $2,000$ queries), our attack recovers over 50% of the queries successfully. If we focus on recovering the characters of the queries instead, our attack achieves over 70% character recovery rate. The queries in the experiments above exclude short queries (queries of length 1 or 2). If such queries are also allowed, the recovery rates reported above increase to 64% and 80% respectively. For the NCBI genome dataset (similar number of characters as $100,000$ Wikipedia pages, $2,000$ queries), our attack

successfully recovers more than 60% of the queries, with a character recovery rate also in excess of 60% .

1.3 Related Work

This section presents a detailed discussion of related work.

Additional Substring-SSE Schemes. To the best of our knowledge, the CS scheme from [14] is the only substring-SSE scheme to rely on suffix trees. There exist only a handful of other substring SSE schemes [20, 36, 51], all of which use very different techniques as compared to suffix trees, and thus have incomparable leakage profiles. Concretely, [20] models substring-matching as conjunctions over n-grams, and uses techniques inspired by SSE for conjunctive keyword queries [9, 10, 38]. On the other hand, [36] treats substring-queries as a special case of range queries, and proposes a construction based on order-preserving encryption [3, 4]. Finally, [51] proposes an approach based on encrypted Ferragina-Manzini indices [22, 23]. The diverse nature of the techniques and leakage profiles for these schemes makes a universal attack strategy against all substring-matching schemes impossible. This is unlike SSE for keyword/range queries, where the vast majority of schemes share similar leakage-profiles (see [33] for a detailed discussion).

FHE-based Substring-Matching. Certain prior works (such as [5]) have explored the usage of fully-homomorphic encryption (FHE) based techniques for substring-matching. The motivation for studying SSE and its leakage is that SSE provides significantly more efficient and scalable alternatives to FHE for specific query classes over structured encrypted databases, albeit at the cost of some leakage to the adversarial server. As a concrete comparison, [20] reports a query-processing time of 40s for their substring-SSE scheme on a 10TB-sized database, while [5] reports a query-processing time of 40s for their FHE-based substring SSE scheme on a single string of length 10 with 1080 characters. Since performance is paramount for real-world applications, studying substring-SSE as an alternative to generic FHE-based solutions is clearly well-motivated. At the same time, understanding the impact of leakage from such SSE schemes is also extremely important. The traditional method for this is leakage-cryptanalysis, which was not studied in the context of substring-SSE prior to our work.

Countermeasures to Leakage Cryptanalysis. Recent works have investigated techniques for suppressing certain kinds of leakage (notably, response volume leakage) from SSE schemes for keyword search [25, 29, 35, 42, 57] and range queries [17]. Since our attack exploits response volume leakage from the CS scheme, a natural question to ask is whether one could generically apply volume leakage suppression to the CS scheme as a countermeasure against our attack. Unfortunately, this does not work as the volume leakage that we exploit in our attack is, in fact, crucial to the query execution methodology of the CS scheme (the leakage arises from a value stored in each node of the suffix tree that is used to determine to how the suffix tree is traversed during query execution – see Section 3 for a more detailed exposition), and it is unclear how the scheme would even function once this leakage is suppressed. We leave it as an interesting open question to design countermeasures against our proposed attack.

2 PRELIMINARIES

In this section, we introduce notations used throughout the paper. We also formally define substring-SSE. For the sake of completeness, we first introduce a syntax for plaintext substring matching. We then present the substring-SSE definition. The definition is in the *single-client, single-server* setting, where the server is assumed to be passively corrupt (i.e., *semi-honest*). We note that this is the most widely studied setting in the SSE literature [9, 10, 13, 15].

Plaintext Substring Matching. Databases for plaintext substring matching are modelled as follows. Let $\mathbf{DB} = (S_1, \ldots, S_N)$ be a collection of strings (which we refer to as a database) over a shared alphabet Σ, i.e. $S_i \in \Sigma^*$ for all $i \in [1, N]$. A plaintext substring query $\mathbf{Query} : \Sigma^* \times (\Sigma^*)^* \rightarrow \{0,1\}^*$ takes as input a plaintext query string $q \in \alpha^*$ and the plaintext database $\mathbf{DB} = (S_1, \ldots, S_N)$, and outputs a list of pairs of indices $((i_k, j_k))_{k=1}^{l}$ such that

$$S_{i_k}[j_k, (j_k + |q|)] = q \quad \forall k = 1, \ldots, l.$$

We say that **Query** is correct if $\mathbf{Query}(\mathbf{DB}, q)$ returns all matching indices of q in **DB** for all **DB** and q. For simplicity, we write $\mathbf{DB}(q)$ to mean $\mathbf{Query}(\mathbf{DB}, q)$.

Syntax of Substring-SSE. A substring-searchable symmetric encryption (substring-SSE) scheme Π is defined by a tuple of three algorithms $\Pi = (\mathbf{Gen}, \mathbf{Setup}, \mathbf{EQuery})$:

- $\mathsf{sk} \leftarrow \mathbf{Gen}_{\mathbf{Clt}}(1^\lambda)$ is a probabilistic algorithm run by the client **Clt**. It takes as input a security parameter 1^λ and outputs a secret key sk.
- $(\perp, \mathbf{EDB}) \leftarrow [\mathbf{Setup}_{\mathbf{Clt}}(\mathsf{sk}, \mathbf{DB}), \mathbf{Setup}_{\mathbf{Svr}}()]$ is an interactive protocol between the client **Clt** and the server **Svr**. The client takes as input a secret key sk and a database **DB** and the server does not take any input. After the interaction, the server outputs an encrypted database **EDB**.
- $(((i_k, j_k))_{k=1}^{l}, \perp) \leftarrow [\mathbf{EQuery}_{\mathbf{Clt}}(\mathsf{sk}, q), \mathbf{EQuery}_{\mathbf{Svr}}(\mathbf{EDB})]$ is an interactive protocol between the client **Clt** and the server **Svr**. The client takes as input a secret key sk and a substring query q and the server takes as input the encrypted database **EDB**. After the interaction, the client obtains a list of responses $((i_k, j_k))_{k=1}^{l}$ and the server obtains no output.

A substring-SSE scheme should satisfy certain correctness and security properties, as defined below.

Correctness of Substring-SSE. We say that a substring-SSE scheme Π is correct if **EQuery** returns the correct substring matching results. That is, for any database **DB** and any query q, when executing the following sequence:

(1) $\mathsf{sk} \leftarrow \mathbf{Gen}_{\mathbf{Clt}}(1^\lambda)$
(2) $(\perp, \mathbf{EDB}) \leftarrow [\mathbf{Setup}_{\mathbf{Clt}}(\mathsf{sk}, \mathbf{DB}), \mathbf{Setup}_{\mathbf{Svr}}()]$
(3) $(((i_k, j_k))_{k=1}^{l}, \perp) \leftarrow [\mathbf{EQuery}_{\mathbf{Clt}}(\mathsf{sk}, q), \mathbf{EQuery}_{\mathbf{Svr}}(\mathbf{EDB})]$

we have $((i_k, j_k))_{k=1}^{l} = \mathbf{Query}(\mathbf{DB}, q)$.

Security of Substring-SSE. We define simulation-based security of substring-SSE against a semi-honest adversarial server (i.e., the server follows the specification of the scheme, but tries to learn information about the underlying plaintext database and queries). Our definition is in the real-world, ideal-world paradigm, and

closely follows the traditional simulation-based security definitions that are widely used in the SSE literature [9, 10, 13, 15]. Let $\Pi = (\mathbf{Gen}, \mathbf{Setup}, \mathbf{EQuery})$ be a substring-SSE scheme. Consider the following probabilistic experiments where $\mathcal{A}$ is a stateful probabilistic polynomial-time (PPT) adversary, $\mathcal{S}$ is a PPT simulator, and $\mathcal{L} = (\mathcal{L}_{\mathbf{Setup}}, \mathcal{L}_{\mathbf{EQuery}})$ is a stateful leakage function:

- $\mathbf{Real}_{\Pi,\mathcal{A}}(1^\lambda)$: the challenger begins by running $\mathbf{Gen}_{\mathbf{Clt}}(1^\lambda)$ to generate a secret key sk. The adversary $\mathcal{A}$ outputs a database $\mathbf{DB}$. The challenger and the adversary $\mathcal{A}$ interact to output $(\bot, \mathbf{EDB}) \leftarrow [\mathbf{Setup}_{\mathbf{Clt}}(\mathsf{sk}, \mathbf{DB}), \mathbf{Setup}_{\mathbf{Svr}}()]$, where the challenger plays the client and $\mathcal{A}$ plays the server. The adversary $\mathcal{A}$ then makes a polynomial number of adaptive substring matching queries: each query involves running $[\mathbf{EQuery}_{\mathbf{Clt}}(\mathsf{sk}, q), \mathbf{EQuery}_{\mathbf{Svr}}(\mathbf{EDB})]$ on an adversarially chosen q, with the challenger acting as the client $\mathbf{Clt}$ and $\mathcal{A}$ acting as the server. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.
- $\mathbf{Ideal}_{\Pi,\mathcal{A},\mathcal{S}}(1^\lambda)$: The adversary $\mathcal{A}$ outputs a database $\mathbf{DB}$. The simulator $\mathcal{S}$ and the adversary $\mathcal{A}$ interact to output $(\bot, \mathbf{EDB}) \leftarrow [\mathcal{S}(\mathcal{L}_{\mathbf{Setup}}(\mathbf{DB})), \mathbf{Setup}_{\mathbf{Svr}}()]$, where $\mathcal{S}$ plays the client and $\mathcal{A}$ plays the server. The adversary $\mathcal{A}$ then makes a polynomial number of adaptive substring matching queries: each query involves running the *simulated* query protocol $[\mathcal{S}(\mathcal{L}_{\mathbf{EQuery}}(q)), \mathbf{EQuery}_{\mathbf{Svr}}(\mathbf{EDB})]$ on an adversarially chosen q, with the simulator $\mathcal{S}$ acting as the client $\mathbf{Clt}$ and $\mathcal{A}$ acting as the server. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.

We say that a substring-SSE scheme Π is $\mathcal{L}$-secure against adaptive chosen-query attacks if for every PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that

$$\left|\mathbf{Pr}[\mathbf{Real}_{\Pi,\mathcal{A}}(1^\lambda) = 1] - \mathbf{Pr}[\mathbf{Ideal}_{\Pi,\mathcal{A},\mathcal{S}}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda).$$

Inference-style Leakage Cryptanalysis. Note that the above security definition informally captures the fact that a semi-honest adversarial server does not learn any information about the client's plaintext database and queries beyond what is captured by the leakage function $\mathcal{L} = (\mathcal{L}_{\mathbf{Setup}}, \mathcal{L}_{\mathbf{EQuery}})$. In this paper, we design query reconstruction attacks where a semi-honest adversary exploits the leakage function $\mathcal{L}$ to recover the client's queries. Our attack is an *inference attack*. More specifically, we assume that the attacker has access to some auxiliary statistical information about the underlying plaintext database. Given this statistical information about the database and the transcript of encrypted query executions, the semi-honest attacker attempts to infer the underlying plaintext queries.

Semi-Honest vs Malicious Corruptions. We remark here that Chase and Shen [14] originally claimed security of their CS scheme in a *strictly stronger* model of security for substring-SSE where the server is allowed to be *maliciously* corrupt. In the security definitions above, we only consider *semi-honest* corruptions of the server, which is a weaker adversarial model (besides this, the security definition above is the same as that in [14]). We choose to present the security definition for substring-SSE in the semi-honest adversarial model to maintain consistency with our proposed attack setting, which assumes a semi-honest adversary.

3 THE CS SCHEME AND ITS LEAKAGE

In this section, we present an overview of the first substring-SSE scheme CS proposed by Chase and Shen [14]. We also present a discussion on its leakage profile, which we subsequently exploit for our query reconstruction attack.

The rest of this section is organized as follows. In Section 3.1, we give a brief introduction to suffix trees, which is the main data structure underlying the CS scheme. Next, in Section 3.2, we present a description of the CS scheme. Finally, in Section 3.3, we outline the leakage of the CS scheme.

3.1 Suffix Trees

In this section, we present an introduction to suffix trees, the primary data structure used in the CS scheme.

Suffix Tree. Let S be a string of length n. A suffix tree T_S for string S is a tree with the following properties:

- The tree has exactly n leaves.
- Except for the root, each internal node has at least two children.
- Each edge is labelled with a non-empty substring of S.
- No two edges with the same starting node share the same starting character.
- Let N be an internal node or a leaf node of T_S. The string on the path from the root to node N is defined as the concatenation of strings on the edges from the root to node N. Node N stores the starting index of the first instance of the string on the path from the root to node N in S.

From now on, we write $\mathsf{ind}(\mathsf{N})$ to denote the string index stored in node N and $\mathsf{path}(\mathsf{N})$ to denote the string on the path from the root to the node N.

Suffix Tree for Multiple Strings. The suffix tree described above can be generalised to support multiple strings. The resultant data structure is known as a generalised suffix tree [1]. The main difference between a suffix tree and a generalised suffix tree is that in the latter case, the nodes no longer store just the starting indices of the matches. Instead, the nodes store the string indices (i.e. in which strings the substring appears) and the starting positions of the matches. We will still call the content stored in the nodes *index* for convenience. Figure 1 gives an example of a generalised suffix tree for the two strings "hello" and "help".

Although Chase and Shen [14] only described how to use their scheme to search over a single string, their scheme can be extended easily to support searching over multiple strings. In our attack, we consider the version of their scheme that supports multiple strings, but our attacks work just as well for the original single-string version.

Substring Query using Generalised Suffix Trees. We now illustrate how to efficiently perform (plaintext) substring matching using a generalised suffix tree representing multiple strings. Consider a substring query "ell". By traversing the suffix tree (Figure 1) with "ell", we will end on node N_9. Although the node N_9 stores the starting index for the suffix "ello", the index is also the starting index of substring "ell" since "ell" is a prefix of "ello". As a result,

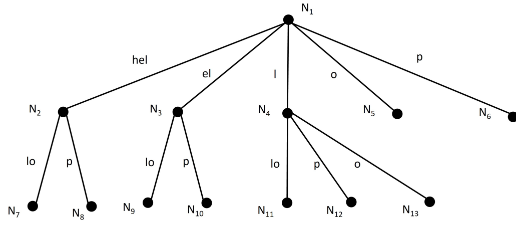


Figure 1: The generalised suffix tree for "hello" and "help". The nodes in the tree store the index of the first occurrence of the string from the root node (N_1) to the current node. For example, N_2 stores $(1, 1)$ since the first occurrence of "hel" occurs at the first position of the first string. The node N_{12} stores $(2, 3)$ since "lp" occurs at the third position in the second string. Some examples for the notations used: $\mathsf{ind}(\mathsf{N}_7) = (1, 1)$; $\mathsf{path}(\mathsf{N}_7)$ = "hello"; $\mathsf{initpath}(\mathsf{N}_9)$ = "ell"; $\mathsf{leafpos}(\mathsf{N}_3) = 3$; $\mathsf{num}(\mathsf{N}_3) = 2$.

we obtain the correct query response corresponding to the query "ell" by traversing the suffix tree.

The suffix tree also supports substring queries with multiple matches. To illustrate that, consider a substring query on "he". Here, instead of retrieving the index stored in node N_2, one can retrieve all the indices stored in the leaf nodes of the subtree rooted at node N_2. Since these leaf nodes contain all the starting indices of suffixes starting with "he", retrieving all indices stored in the leaf nodes yield all matches of "he".

3.2 The CS Scheme: Building Substring-SSE from Suffix Trees

In this section, we show how Chase and Shen [14] built their substring-SSE scheme CS from suffix trees. Following their presentation, we show three attempts at building a substring-SSE scheme.

A First Attempt. The first idea from [14] is as follows. Let F be a pseudorandom function and Γ be a CPA-secure symmetric encryption scheme. Let sk_1 be a key for F and sk_2 be a key for Γ. Let node N be a non-root node of the suffix tree. The substring-SSE scheme encrypts compute a PRF value $t = \mathsf{F}_{\mathsf{sk}_1}(\mathsf{path}(\mathsf{N}))$ and encrypts the index stored in node N as $c = \Gamma.\mathsf{Enc}_{\mathsf{sk}_2}(\mathsf{ind}(\mathsf{N}))$. The scheme stores (t, c) in an encrypted dictionary D where t is used as the key and c is used as the value. This process is performed on all non-root nodes.

This scheme allows the client to query all substrings that are paths in the suffix tree by using encrypted queries of the form $eq = \mathsf{F}_{\mathsf{sk}_1}(q)$. If q is one of the paths, then the PRF value $\mathsf{F}_{\mathsf{sk}_1}(q)$ is in the encrypted dictionary D, and the value stored in $\mathsf{D}[t]$ is an encryption of the index of the first match for q. Note that this scheme only returns the index of the first match since tree traversal is impossible with the encrypted dictionary D.

Returning a Possible Match. Define $\mathsf{initpath}(\mathsf{N})$ as the concatenation of strings on the edges from the root to node N as before, except that for the last edge (closest to N), only the first character of the string on the edge is used. For example, in Figure 1, $\mathsf{initpath}(\mathsf{N}_7)$ = "hell" and $\mathsf{initpath}(\mathsf{N}_9)$ = "ell".

In the second attempt, Chase and Shen make the following changes. Instead of querying node N with $\mathsf{F}_{\mathsf{sk}_1}(\mathsf{path}(\mathsf{N}))$, we query node $\mathsf{initpath}(\mathsf{N})$. This can be done by computing the encrypted query as $eq = (\mathsf{F}_{\mathsf{sk}_1}(q[1, 1]), \ldots, \mathsf{F}_{\mathsf{sk}_1}(q[1, |q|]))$. Given eq, the server finds the PRF value with the longest substring of q in the encrypted dictionary D and returns the value associated with it. The client then decrypts the value to obtain a string index (i, j). After that, the client sends $(i, j|q|)$ to the server, retrieves the following encrypted values (these should be stored on the server in the **Setup** phase):

$$\Gamma.\mathsf{Enc}_{\mathsf{sk}_2}(S_i[j]), \ldots, \Gamma.\mathsf{Enc}_{\mathsf{sk}_2}(S_i[j + |q| - 1]),$$

and checks if the decryptions of the characters are the same as q. If the characters match, then (i, j) is an index for the match. Otherwise, q does not have any match in the database.

Returning All Occurrences. A trivial solution to allow the client to retrieve all matching occurrences of a query is to store all matching indices in the encrypted dictionary. However, this solution has a linear storage blow-up (with respect to the lengths of the strings the server stores) in the worst case.

To overcome this problem, Chase and Shen used the fact that if node N is the node that matches a query in the second attempt, all matching indices of the query must be exactly the indices stored in the leaf nodes in the subtree of N. Hence, it suffices to store the indexing information of the subtree of N in N, so that the leaf nodes can be accessed later.

This is done as follows. Let leaf_i be the i-th leaf node of the suffix tree. In the **Setup** phase, the client creates an encrypted array L where $\mathsf{L}[i] = \Gamma.\mathsf{Enc}_{\mathsf{sk}_2}(\mathsf{ind}(\mathsf{leaf}_i))$. This is the same index as the encrypted dictionary D stores for the leaf nodes in the second attempt. As for the encrypted dictionary D, in addition to the index of the first occurrence, we also store the subtree information for every entry. For node N, define $\mathsf{leafpos}(\mathsf{N})$ as the position of the leftmost leaf node in the subtree of N. Define $\mathsf{num}(\mathsf{N})$ as the number of leaf nodes in the subtree of N.[3] Then, for the entry with dictionary key $\mathsf{F}_{\mathsf{sk}_1}(\mathsf{initpath}(\mathsf{N}))$, we store $\Gamma.\mathsf{Enc}_{\mathsf{sk}_2}(\mathsf{ind}(\mathsf{N}), \mathsf{leafpos}(\mathsf{N}), \mathsf{num}(\mathsf{N}))$.

An encrypted substring query will now proceed as follows. Let q be a query string. The client begins by computing $eq = (\mathsf{F}_{\mathsf{sk}_1}(q[1, 1]), \ldots, \mathsf{F}_{\mathsf{sk}_1}(q[1, |q|]))$ and sending it to the server. The server finds the PRF value with the longest substring of q in the encrypted dictionary D and returns the value associated with it. The client decrypts the value and get $\mathsf{ind}(\mathsf{N}), \mathsf{leafpos}(\mathsf{N}), \mathsf{num}(\mathsf{N})$ for some node N that is a prefix of q. The client retrieves

$$\Gamma.\mathsf{Enc}_{\mathsf{sk}_2}(S_i[j]), \ldots, \Gamma.\mathsf{Enc}_{\mathsf{sk}_2}(S_i[j + |q| - 1]),$$

and checks if the decryptions of the characters are the same as q. If they are the same, the client retrieves

$$\mathsf{L}[\mathsf{leafpos}(\mathsf{N})], \ldots, \mathsf{L}[\mathsf{leafpos}(\mathsf{N}) + \mathsf{num}(\mathsf{N}) - 1],$$

and decrypts them to obtain all indices of matching occurrences. Otherwise, the client knows that there are no matching occurrences for query q.

Further Modifications. The final CS scheme makes the following modifications to the third attempt to reduce its leakage:

[3] See Figure 1 for an example.

- The search tokens are encrypted and the content of the encrypted dictionary is modified so that the scheme only leaks information when two queries have a shared prefix.
- Node degrees and the number of nodes in the suffix tree are obfuscated by padding. The order of children nodes are hidden by permuting them.
- The string indices are hidden by permuting the encrypted ciphertexts, i.e. $\Gamma.\mathsf{Enc}_{\mathsf{sk}_2}(S_1[1]), \ldots, \Gamma.\mathsf{Enc}_{\mathsf{sk}_2}(S_N[|S_N|])$.

3.3 Leakage Profile of the CS Scheme

We now describe the leakage profile of the CS scheme. We include an example to illustrate the leakage.

Leakage Description. During **Setup**, the scheme leaks the total length of the strings. That is, if $\mathbf{DB} = (S_1, \ldots, S_N)$, then we have

$$\mathcal{L}_{\mathbf{Setup}}(\mathbf{DB}) = \sum_{i=1}^{N} |S_i|.$$

We now focus on the leakage during queries, i.e., $\mathcal{L}_{\mathbf{EQuery}}$. For a substring query q, let $\mathsf{initpath}(\mathsf{N})$ where N is a node in the suffix tree T be the longest prefix of q. When processing a query q, the scheme leaks the length of the query $|q|$ and the length of the prefix $|\mathsf{initpath}(\mathsf{N})|$. For a sequence of queries $q_1, \ldots, q_l$, the CS scheme leaks three additional patterns:

- *The query prefix pattern* $\mathsf{QP}(\mathbf{DB}, q_1, \ldots, q_l)$ for query q_l indicates for every node visited by query q_l, if the node is visited by any of the previous queries. The pattern can be represented by an $l \times n_i$ matrix, where n_i is the number of nodes visited by query q_l. The (i, j)-th entry of the matrix is 1 if q_i visited the j-th node that is visited by query q_l; the entry is 0 otherwise.
- *The leaf intersection pattern* $\mathsf{LP}(\mathbf{DB}, q_1, \ldots, q_l)$ for query q_l indicates which leaf indices retrieved by query q_l are also retrieved by previous queries. The pattern can be represented by an $l \times m_j$ matrix, where m_j is the number of leaf nodes retrieved by query q_l. Let $r_1 : [m_j] \to [m_j]$ be a random permutation. The (i, j)-th entry of the leaf intersection pattern matrix is 1 if the $r_i(i)$-th leaf retrieved by query q_l is also retrieved by query q_i; the entry is 0 otherwise.
- *The index intersection pattern* $\mathsf{IP}(\mathbf{DB}, q_1, \ldots, q_l)$ for query q_l indicates which string indices retrieved by query q_l are also retrieved by previous queries. The pattern can be represented by an $l \times |q_l|$ matrix. Let $r_2 : [|q_l|] \to [|q_l|]$ be a random permutation. The (i, j)-th entry of the index intersection pattern matrix is 1 if the i-th string index retrieved by query q_l is also retrieved by query q_i; the entry is 0 otherwise.

Leakage Illustration using an Example. To illustrate the leakage of the scheme, consider the database $\mathbf{DB}$ = ("hello", "help"). The server learns that there are 9 leaf nodes from $|\mathsf{L}|$, which is exactly the total length of the strings.

Now, consider query q_1 = "ello" and q_2 = "elp". The server learns that $|q_1| = 4$ and $|q_2| = 3$ since the server sees 4 and 3 encrypted PRF values, respectively. The server also learns that $|(\mathsf{initpath}(\text{"ello"}))| = |\text{"ell"}| = 3$ and $|(\mathsf{initpath}(\text{"elp"}))| = |\text{"elp"}| = 3$ since the PRF values used to retrieve the nodes associated to these initial paths reveal that the inputs to the PRF have length 3. When looking at q_1 and q_2 together, we see that a semi-honest adversary can immediately infer the following:

- *The query prefix pattern:* The server sees node N_3 is a shared prefix for both queries since it is visited by both queries. The server also sees that $|\mathsf{initpath}(\mathsf{N}_3))| = 1$ since it has to be retrieved with $\mathsf{F}_{\mathsf{sk}_1}(q_1[1, 1])$ or $\mathsf{F}_{\mathsf{sk}_1}(q_2[1, 1])$.
- *The leaf intersection pattern:* The server does not see any leaf intersection since the two queries visit disjoint sets of nodes ($\{\mathsf{N}_9\}$ and $\{\mathsf{N}_{10}\}$ respectively). However, the server can infer that both queries have query response volume 1.
- *The index intersection pattern:* The server does not see any index intersection since the first string is on the first occurrence of $\mathsf{initpath}(q_1)$ and the second string is on the first occurrence of $\mathsf{initpath}(q_2)$.

4 OUR QUERY RECOVERY ATTACK

In this section, we describe our query reconstruction attack against the CS scheme. We begin by highlighting the main leakage components that we exploit in our attack. We then introduce the general idea of our attack and present its pseudocode.

4.1 Notable Leakage Components

Our attack exploits the following leakage components of the scheme.

- *Length of the initial path:* Our attack aims to recover the initial paths of the queries. The length of the initial path, or $\mathsf{ipLen}_i = |\mathsf{initpath}(q_i)|$ is useful for the attacker to restrict the set of guesses it can make on the initial path.
- *Query response volume:* The query response volume, or $\mathsf{vol}_i = |\mathbf{DB}(\mathsf{initpath}(q_i))|$ can be inferred from the leaf intersection pattern. This allows an attacker to extract frequency information about the initial path in the database.
- *Character equality between the queries:* From the leaf intersection pattern, the attacker can find out if two queries start with the same characters. This can be done as follows. Consider queries q_1 ="ell" and q_2 ="elp" on the suffix tree in Figure 1. The attacker learns that both queries visit N_3. This is only possible if the first two characters of the two queries are the same. Note that the number of shared characters is leaked because the length of the initial paths of the two queries is one more than the string on the path of N_3. We write $\mathsf{charEq}(i, j) = k$ to mean that q_i and q_j have k common characters. In the example above, $\mathsf{charEq}(1, 2) = 2$. Character equality between the queries can be used by the attacker to refine its guesses for the queries.

Leakage Extraction. The above leakage components can be derived directly from the transcript of an execution of the CS scheme. We show how this can be done below.

Extracting the Length of the Initial Path. Recall that for query q, the client sends $\mathsf{F}_{\mathsf{sk}_1}(q[1, 1]), \ldots, \mathsf{F}_{\mathsf{sk}_1}(q[1, |q|])$ to the server. The server finds the largest m such that $\mathsf{F}_{\mathsf{sk}_1}(q[1, m])$ is in the encrypted database. Since m is equal to the length of the initial path by construction, the server can learn the length of the initial path from the transcript.

Extracting the Query Response Volume. In the last step of the search, suppose that the longest initial path matching the query is N. Then, the client will retrieve L[leafpos(N)], ..., L[leafpos(N)+num(N)−1] (a permuted version of this is used in the final scheme). The number of leaf nodes retrieved is equal to num(N), i.e. the the number of indices matching N. This allows the server to learn the query response volume directly.

Extracting Character Equality across Queries. For simplicity, consider queries q_i and q_j. For q_i, the client computes and sends to the server the PRF values $\mathsf{F}_{\mathsf{sk}_1}(q_1[1,1]), \ldots, \mathsf{F}_{\mathsf{sk}_1}(q_1[1,|q|])$. For q_2, these values are $\mathsf{F}_{\mathsf{sk}_1}(q_j[1,1]), \ldots, \mathsf{F}_{\mathsf{sk}_1}(q_j[1,|q|])$. Crucially, if q_i and q_j have a non-empty common prefix, i.e. there exists $k \geq 1$ such that $q_i[1,k] = q_j[1,k]$, then the first k PRF values from the two queries will be the same. As a result, the server can learn the character equality between the queries just by comparing the PRF values sent by the client.

Leakage Representation. In our attack, we process the leakage described above and represent the encrypted queries as lists of tokens where each token is an integer. If the same token appears in two encrypted queries, it means the corresponding characters in the two queries are the same. For example, for q_1 ="ell" and q_2 ="elp", since $\mathsf{charEq}(1,2) = 2$, we can represent the encrypted version of the queries as $(1,2,3)$ and $(1,2,4)$ respectively. Our attack then tries to map the tokens to the alphabets used in the queries. It is worth noting that the encrypted queries we are interested in are only as long as the initial paths of the original queries. This is because the server can only learn information about the initial path by design. We use the following notations in the description of our attack below. We abuse the notation and write $eq_i = (\mathsf{tk}_{i,1}, \ldots, \mathsf{tk}_{i,m_i})$ where m_i is the length of the initial path of query q_i. We write $|\mathbf{DB}(eq_i))|$ to mean the query response volume of (the initial path of) query q_i. Using the notation above, the leakage input to our attack is simply the collection of sequences of tokens $(\mathsf{tk}_{i,1}, \ldots, \mathsf{tk}_{i,m_i})_{i=1}^{l}$ where l is the number of queries made, and their corresponding query response volumes $(\mathsf{vol}_i)_{i=1}^{l}$ where $\mathsf{vol}_i = \left|\mathbf{DB}((\mathsf{tk}_{i,1}, \ldots, \mathsf{tk}_{i,m_i}))\right|$.

4.2 Our Attack

Given the leakage above, we design an inference attack that tries to recover the alphabets corresponding to the tokens. By doing so, we recover the initial path of the queries.

Main Ideas. There are three core components in our attack.

- First of all, we need to identify the set of guesses we want to make on each encrypted query. We call this the *candidate set* of the encrypted query. On a high level, given some auxiliary information, we can use the length of the initial path and the query response volume to filter out unlikely guesses for each encrypted query. Then, we can use the character equality leakage between the queries to reduce the size of the candidate sets further.
- Secondly, once we obtain the candidate sets for all the encrypted queries, we need to have a statistical model to measure how well guesses on the encrypted queries fit the observed leakage. For this, we model the number of occurrences of the substrings with a Poisson distribution, where the parameters of the distribution are determined by some auxiliary data. Then, we compute the *likelihood* of a guess given the observed leakage components.
- Finally, we need to use an efficient algorithm to search over the set of possible guesses and find the most likely one given the statistical model.

Identifying the Candidate Sets. Define $\mathsf{CandSet} : \mathbb{N}* \rightarrow \mathcal{P}(\Sigma^*)$ as a map that maps a sequence of tokens to a subset of all possible strings. Our goal is to find candidate sets of each encrypted query. We aim to make the candidate sets as small as possible so that the search procedure later will be as efficient as possible.

In our attack, we identify the candidate sets in two steps. In the first step, we rely on frequency analysis. Suppose that there is some auxiliary information $\mathsf{Aux} : \Sigma^* \rightarrow \mathbb{R}$ such that $\mathsf{Aux}(s)$ tells the attacker the expected query response volume of query s. The attacker can simply set $\mathsf{CandSet}((\mathsf{tk}_{i,1}, \ldots, \mathsf{tk}_{i,m_i}))\{s : |s| = m_i \wedge \left|\left|\mathbf{DB}((\mathsf{tk}_{i,1}, \ldots, \mathsf{tk}_{i,m_i}))\right| - \mathsf{Aux}(s)\right| < \varepsilon \cdot \sqrt{\left|\mathbf{DB}((\mathsf{tk}_{i,1}, \ldots, \mathsf{tk}_{i,m_i}))\right|}\}$ for some threshold ε. This is exactly what we do in our attack. Looking ahead, we describe in Section 5 how the threshold ε is determined in our experiments over real-world datasets.

However, the problem with the step above is that the candidate sets produced are often very large (on the order of 10^3 in size or more). For l queries, the search space is then roughly 10^{3l} in size. This makes efficient search over the space infeasible. As a result, we introduce a procedure to trim the search space before running the main algorithm of our attack. The idea of the trimming algorithm is that we can look at the candidate sets for each token (instead of the whole token sequence) by making use of the candidate sets and determine the most likely characters for each token. This can then be used to trim the candidate sets.

As an example, consider the following candidate sets. For token sequence $(1,2,3)$, $\mathsf{CandSet}((1,2,3)) = \{$"ell", "ali"$\}$. And for token sequence $(1,2,4)$, $\mathsf{CandSet}((1,2,4)) = \{$"elp", "bob"$\}$. From these, we can conclude that the character corresponding to token 1 is most likely "e" since it appears in both candidate sets. This then allows us to trim both of the candidate sets to $\mathsf{CandSet}((1,2,3)) = \{$"ell"$\}$ and $\mathsf{CandSet}((1,2,4)) = \{$"elp"$\}$ respectively. One complication here is that it is possible that the true query may not be included in the candidate set (because the observed query response volume is too far from the expected frequency of the substring). For this reason, we opt for a soft decision procedure where characters that do not appear in all candidate sets for a particular token may still be regarded as likely. In the paragraph below, we describe more formally how the trimming process is done.

We initialise an array $Arr : \mathbb{N} \times \Sigma \rightarrow \mathbb{N}$ that maps tokens and characters in the alphabet to an integer. This array is initialised with all possible token-character pairs and the values are set to zero. Then, for every token sequence $(\mathsf{tk}_{i,1}, \ldots, \mathsf{tk}_{i,m_i})$ we observe, we tally the characters that appear at each position in the candidate set $\mathsf{CandSet}((\mathsf{tk}_{i,1}, \ldots, \mathsf{tk}_{i,m_i}))$. For every token tk in the token sequence and every character c that appears in the corresponding positions, we increment $Arr(\mathsf{tk}, c)$ by 1. After this process is completed for all token sequences, we create a map $\mathsf{TokenCandSet} : \mathbb{N} \rightarrow \mathcal{P}(\Sigma)$ that maps each token to a subset of the alphabet. The entries of TokenCandSet are such that for token tk, $\mathsf{TokenCandSet}(\mathsf{tk})$ is set to the collection of characters c such

Algorithm 1 Identifying Candidate Sets

Parameters: Parameter for the initial identification of the candidate sets ε, parameter for trimming the candidate sets t.
Input: Query response volumes $(\mathsf{vol}_i)_{i=1}^{l}$, sequences of tokens extracted from the encrypted queries $(\mathsf{tk}_{i,1},\dots,\mathsf{tk}_{i,m_i})_{i=1}^{l}$, auxiliary distribution Aux.
Output: Candidate set CandSet : $\mathbb{N}^* \to \mathcal{P}(\Sigma^*)$ that maps each sequence of tokens to a set of plaintext strings.

```
Candidate_Set((vol_i)_{i=1}^l, (tk_{i,1},...,tk_{i,m_i})_{i=1}^l, Aux)
1: Initialise CandSet as an empty map
2: for i ← 1,...,l do
3:     CandSet((tk_{i,1},...,tk_{i,m_i})) ← {s ∈ Aux | |s| = m_1 ∧ |vol_i − Aux(s)| ≤
   ε · √vol_i}   ▷ Initial filtering of candidate sets based on query response volumes.
   s ∈ Aux means getting the strings used to index Aux.
4: CandSet′ ← CandSet
5: CandSet ← Trim_Candidates(CandSet)
6: while CandSet′ ≠ CandSet do
7:     CandSet′ ← CandSet
8:     CandSet ← Trim_Candidates(CandSet)
9: return CandSet

10: Trim_Candidates(CandSet)
11: Initialise Arr : ℕ × Σ → ℕ as an array filled with zeros
12: for i ∈ 1,...,l do
13:     for j ∈ 1,...,m_i do
14:         for c ∈ Σ do
15:             if ∃ cand ∈ CandSet((tk_{i,1},...,tk_{i,m_i})) such that cand[j] = c then
16:                 Arr(tk_{i,j}, cand[j]) ← Arr(tk_{i,j}, cand[j]) + 1
17: Initialise TokenCandSet : ℕ → 𝒫(Σ) as an empty map
18: for tk ∈ tk_{1,1},...,tk_{l,m_l} do
19:     Let c_1,...,c_k be the second input of Arr sorted by decreasing Arr(tk, ·)
20:     TokenCandSet(tk) ← {c_1,...,c_t}
21:     if Arr(tk, c_t) = Arr(tk, c_{t+1}) then ▷ Handling the special case where more
    than t characters have the same frequency in Arr(tk, ·)
22:         j ← t
23:         while Arr(tk, c_j) = Arr(tk, c_{j+1}) do
24:             TokenCandSet(tk) ← TokenCandSet(tk) ∪ {c_{j+1}}
25: CandSet′ ← {}
26: for (tk_1,...,tk_l) ∈ CandSet do
27:     CandSet(cand) = {(c_1,...,c_l) ∈ CandSet((tk_1,...,tk_l)) | c_i ∈
    TokenCandSet(tk_i) ∀i}
28: return CandSet′
```

Algorithm 2 Simulated Annealing-based Attack Procedure

Input: Query response volumes $(\mathsf{vol}_i)_{i=1}^{l}$, sequences of tokens extracted from the encrypted queries $(\mathsf{tk}_{i,1},\dots,\mathsf{tk}_{i,m_i})_{i=1}^{l}$, candidate set CandSet, auxiliary distribution Aux, the maximum number of iterations for simulated annealing $\mathsf{iter}_{\max}$.
Output: A guess for the tokens of the encrypted queries guess : $\mathbb{N} \to \Sigma$.

```
Simulated_annealing((vol_i)_{i=1}^l, (tk_{i,1},...,tk_{i,m_i})_{i=1}^l, CandSet, Aux, iter_max):
1: guess ← Initial_solution(CandSet)
2: for i ← 1,...,iter_max do
3:     T ← Cooling(T)
4:     guess′ ← Neighbour(guess, CandSet)
5:     sc ← score(guess, (tk_{i,1},...,tk_{i,m_i})_{i=1}^l, (vol_i)_{i=1}^l, Aux)
6:     sc′ ← score(guess′, (tk_{i,1},...,tk_{i,m_i})_{i=1}^l, (vol_i)_{i=1}^l, Aux)
7:     if Accept_prob(sc, sc′, T) == 1 then
8:         guess ← guess′
9: return guess

Initial_solution(CandSet):
10: guess ← {}
11: for (tk_{i,1},...,tk_{i,m_i}) ∈ CandSet do   ▷ t ∈ CandSet means getting the keys of
    the map.
12:     cand ←$ CandSet((tk_{i,1},...,tk_{i,m_i}))
13:     for j ∈ 1,...,m_i do
14:         guess(tk_{i,j}) ← cand[j]
15:     return guess

Neighbour(guess, CandSet):
16: guess′ ← guess
17: (tk_{i,1},...,tk_{i,m_i}) ←$ CandSet   ▷ t ←$ CandSet means sampling a random key
    from the map.
18: cand ←$ CandSet((tk_{i,1},...,tk_{i,m_i}))
19: for j ∈ 1,...,m_i do
20:     guess′(tk_{i,j}) ← cand[j]
21: return guess′

Accept_prob(sc, sc′, T):
22: return exp{(sc′−sc)/T} > rand(0,1)   ▷ rand(0,1) is an uniform random variable
    with range from 0 to 1.
```

that the values $Arr(\mathsf{tk}, c)$ are the t largest among $Arr(\mathsf{tk}, \cdot)$. If more than t characters share the largest value, then all characters with the largest value in $Arr(\mathsf{tk}, \cdot)$ are added to TokenCandSet(tk). The map TokenCandSet can then be used to trim CandSet. This process is repeated until CandSet does not change by the trimming process. The pseudocode of the trimming process can be found in Algorithm 1.

Statistical Model of the Observed Leakage. Given the candidate set CandSet, the attacker can start making guesses guess : $\mathbb{N}^* \to \Sigma^*$ on the plaintexts of the initial paths of the queries. However, we still need to measure how good each guess is so that we can pick the best one as the output of the attack. To do this, we model the query response volumes of the queries using Poisson distributions. These individual distributions are put together to form a joint distribution over all queries. We then apply standard statistical techniques to turn the joint distribution into a likelihood function.

The Likelihood Function. Due to the limitation on space, we present the derivation of the likelihood function in Appendix A.

In the rest of this section, we describe how the attacker can search for a guess that maximizes the likelihood function.

Consistency in the Guesses. Note that in the description of our attack above, guesses are made on each query separately. This means it is possible to have guess((1, 2, 3)) = "elp" and guess((1, 2, 4)) = "bob" even though we know the two queries must have the same first two characters. In our attack, we choose to make guesses iteratively and keep track of the guesses we make on the individual tokens. A new guess on a token is allowed to overwrite an old guess. Concretely, for the example above, we will make the guess guess((1, 2, 3)) = "elp" first followed by guess((1, 2, 4)) = "bob". Since the second guess overwrites the guesses for token 1 and 2, it changes guess((1, 2, 3)) to "bop". This approach allows us to maintain consistency of token equality at the cost of potentially sub-optimal guesses for the queries. We believe that this is not a major issue as the sub-optimal guesses yield low likelihood scores and they are unlikely to be accepted by the optimization algorithm we describe subsequently.

Maximizing the Likelihood Function. We use simulated annealing [61] to maximize the likelihood function. The corresponding pseudocode can be found in Algorithm 2 (the score function **score**() is precisely the likelihood function, which was described mathematically above). Th procedure involves five main subroutines:

- **Initial_solution**(CandSet): The algorithm takes as input the candidate set CandSet and outputs a guess guess which will be used as the initial solution of simulated annealing.
- **Neighbour**(guess, CandSet): The algorithm takes as input the current guess guess and the candidate set CandSet, and output a new guess guess′ that is close to guess.
- **score**(guess, $(\mathsf{tk}_{i,1}, \dots, \mathsf{tk}_{i,m_i})_{i=1}^{l}$, $(\mathsf{vol}_i)_{i=1}^{l}$, Aux): The algorithm takes as input a guess guess, a list of tokens $(\mathsf{tk}_{i,1}, \dots, \mathsf{tk}_{i,m_i})_{i=1}^{l}$, a list of observed query response volumes $(\mathsf{vol}_i)_{i=1}^{l}$, and an auxiliary distribution Aux, and output a score sc.
- **Cooling**(T): The algorithm takes as input a temperature T and outputs a new temperature T′.
- **Accept_prob**(sc, sc′, T): The algorithm takes as input two scores sc and sc′ and a temperature T, and outputs 0 or 1 depending on the inputs and some randomness.

The five subroutines are combined as follows in a full run of simulated annealing:

- First, guess ← **Initial_solution**(CandSet) is run to get an initial guess guess.
- Next, a temperature T is initialised.
- Subsequently, the following procedure is executed iteratively for a fixed number of iterations:
  - The cooling subroutine **Cooling**(T) is invoked to obtain a new temperature T′. A neighbour guess′ ← **Neighbour**(guess, CandSet) of the current guess is also computed.
  - Then, a score sc′ ← **score**(guess′, $(\mathsf{tk}_{i,1}, \dots, \mathsf{tk}_{i,m_i})_{i=1}^{l}$, $(\mathsf{vol}_i)_{i=1}^{l}$, Aux) is computed on the guess. Assume that the old score sc ← **score**(guess′, $(\mathsf{tk}_{i,1}, \dots, \mathsf{tk}_{i,m_i})_{i=1}^{l}$, $(\mathsf{vol}_i)_{i=1}^{l}$, Aux) is kept by simulated annealing, it can call **Accept_prob**(sc, sc′, T) to get an output 0 or 1.
  - If the output is 0, then nothing happens, and simulated annealing proceeds to the next iteration.
  - If the output is 1, the old guess guess is overwritten by the new guess guess′, and simulated annealing proceeds to the next iteration.

5 EXPERIMENTAL EVALUATION

In this section, we present extensive experimental evaluations to validate the practicality of our proposed attack. We use two real-world datasets – English Wikipedia[4], and a genome dataset[5] from the National Center for Biotechnology Information (NCBI). A description of these datasets can be found in Appendix B. Our experiments show that our attack achieves a high success rate with reasonable practical efficiency while scaling to large datasets. Our attack code is publicly available on Github.[6]

5.1 Experiments on Simple English Wikipedia

We now present the experimental results for our attack with the Simple English Wikipedia as target dataset.

Experimental Data and Auxiliary Information. From the dataset, we randomly pick 100, 000 Wikipedia pages as the auxiliary dataset. From the remaining Wikipedia pages, we picked from 20, 000 to 100, 000 Wikipedia pages in steps of 20, 000 pages as the target dataset.

We computed Aux by building a suffix tree with the auxiliary dataset. For every initial path initpath in the suffix tree, we assign Aux(initpath) = vol, where vol is the number of times initpath appears in the auxiliary dataset. For each attack we run, Aux is normalised by dividing by the total length of the texts in the auxiliary dataset and multiplying by the total length of the texts in the target dataset. For the target datasets, we build suffix trees and extract leakage using the process described below.

Attack Parameters. We now discuss how we picked the attack parameters for our attack on the Simple English Wikipedia. We begin by recalling the parameters ε and t.

Parameters. Parameter ε is used to control the number of substrings we add to the candidate set for each query in the initial identification of the candidate sets. A larger ε means it is more likely for the candidate sets to include the correct guess. On the other hand, a larger ε also makes the candidate sets larger. This reduces the effectiveness of simulated annealing as it becomes harder to pick the correct guesses from the candidate sets.

Parameter t is used to trim the candidate sets. A larger t means we are more forgiving in the trimming step, so it is more likely for the correct guess to remain in the candidate set. On the other hand, a larger t is not so effective in reducing the candidate sets, thus, making it harder for the simulated annealing step to find the correct guesses.

Considerations. In summary, there are two conflicting considerations in parameter selections. The first consideration is the sizes of the candidate sets. We want them to be as small as possible. In our experiments, we report the product of the sizes of the candidate sets as the size of the reconstruction space (in $\log_{10}$). The second consideration is how often are the correct guesses captured in the candidate sets. We want this to be as often as possible. In our experiments, we report the fraction of the candidate sets including the correct guesses as the hit rate.

Experimental Results. We pick ε from 3 to 7 and t from 3 to 7 in our experiments (with 100, 000 strings and 10, 000 queries). In addition, we also run experiments without using a threshold t. A selection of our experimental results are shown in Table 2 in Appendix C.

There are two interesting observations. Firstly, the size of the reconstruction space does not decrease monotonically with increasing ε. For example, with $\varepsilon = 3, t = 3$, the reconstruction space has size $10^{15039.46}$, whereas with $\varepsilon = 7, t = 3$, the reconstruction space has size $10^{7757.57}$. This is different from what one intuitively expects. However, it can be explained by the trimming step of our attack. When ε is too small, the candidate sets do not contain enough correct guesses for the trimming step to be effective, and that leads to significantly larger reconstruction spaces.

Secondly, the threshold t is very effective at reducing the size of the reconstruction space without sacrificing the hit rate. For example, the reconstruction space for $\varepsilon = 7$ and no trimming is $10^{10787.98}$ and the hit rate is 91.32%. By trimming with $t = 3$, we reduce the search space by a factor of 10^{3030} while only sacrificing 0.42% of the hit rate.

[4]https://dumps.wikimedia.org/simplewiki/
[5]https://www.ncbi.nlm.nih.gov/
[6]https://github.com/substring-SSE-attack/substring-attack

Chosen Parameters. Out of all combinations of parameters we have tested, $\varepsilon = 7, t = 3$ achieves the best balance between the size of the reconstruction space and the hit rate. So this setting is used in our main attack against the Simple English Wikipedia dataset.

Query Generation. Queries in our experiments are generated randomly. For each query, we randomly pick a Wikipedia page from the target dataset. Then, we pick a random initial position inside the Wikipedia page. The final position is picked by generating a random number (specified later) and adding it to the initial position. The plaintext query is then the substring specified by the initial position and the final position in the randomly picked Wikipedia page. There are several conditions the plaintext query must meet before being consumed. These conditions are: (1) the query must have a specific length (to be specified later), (2) the query response volume must be larger than 100 (this is to avoid using erroneously extracted plaintexts or uncommon short strings as queries; an example of that would be "== == Other" in the Simple English Wikipedia article on Alan Turing[7]), and (3) the letters in the queries should only contain English letters, space and hyphen.

We generate 10,000 random queries for the target datasets with 20,000 to 80,000 Wikipedia pages. For the target with 100,000 Wikipedia pages, we generate 2,000 to 10,000 random queries.

Metrics. We use four metrics to measure the attack success rate:

- **Unique token recovery rate**: The percentage of unique tokens our attack recovers correctly. For example, if our attack correctly recovers tokens $(1, 2, 3)$ as "elp" and wrongly recovers tokens $(1, 2, 4)$ as "elt", the unique token recovery rate will be $3/4 = 75\%$, since tokens 1 and 2 are only counted once.
- **Token recovery rate with repetition**: The percentage of tokens (with repetition) our attack recovers correctly. For example, if our attack correctly recovers tokens $(1, 2, 3)$ as "elp" and wrongly recovers tokens $(1, 2, 4)$ as "elt", the unique token recovery rate will be $5/6 = 75\%$. This metric reflects the percentage of characters we can guess correctly per query on average.
- **Initial path recovery rate**: The percentage of the initial paths of the queries our attack recovers correctly. This is equivalent to the percentage of queries for which we can guess all of the tokens correctly. For example, if our attack correctly recovers tokens $(1, 2, 3)$ as "elp" and wrongly recovers tokens $(1, 2, 4)$ as "elt", the initial path recovery rate will be $1/2 = 50\%$.
- **Query recovery rate**: The percentage of queries recovered correctly.

Attack Experiments. We present three sets of experimental results against the CS scheme. In these experiments, we investigate how well our attack scales with the number of queries and with the number of strings.

For the first set of experiments, we use 100,000 Wikipedia pages as the target dataset and 10,000 queries to generate the leakage. The lengths of the queries are specified in Table 1. We observe that the attack performs the best when queries with lengths 1 and 2 are allowed. Naturally, as queries with lengths 1 and 2 have the largest query response volumes, they are easier to recover. More interestingly, these short queries also help with the query recovery rate of longer queries. In particular, in our experiment with query lengths from 1 to 11, the initial path recovery rate for the queries with lengths from 3 to 11 is 65.1%. This is 13.1% higher than in the attack with only queries with lengths from 3 to 11. The reason for this improvement is that the queries with lengths 1 and 2 have small candidate sets (since they have the largest query response volumes which made them easy to identify). Since these short queries share the same initial paths (and hence, the tokens in our attack) with the longer queries, the smaller candidate sets for the short queries also help to reduce the size of the candidate sets for the longer queries, and ultimately lead to a higher query recovery rate.

Query Length	Recovery Metrics			
	Unique Token	Token with Repetition	Initial Path	Query
1-11	60.1%	81.9%	66.3%	63.6%
3-7	56.4%	75.5%	57.3%	56.6%
3-9	53.5%	74.2%	55.5%	54.1%
3-11	51.9%	71.6%	52.0%	49.8%
3-13	51.8%	72.3%	53.2%	50.4%

Table 1: Performance of our attack on Simple English Wikipedia with respect to the query length (indicated by the first column). ***a*-*b*** **in the first column means that the query length is uniformly picked between** ***a*** **and** ***b*** **(inclusive).**

For the second set of experiments, we use 100,000 Wikipedia pages as the target dataset and use 2,000 to 10,000 queries (with lengths between 3 and 11) to generate the leakage. The results are shown in Figure 2a. The unique token reconstruction rate, the initial path reconstruction rate and the query reconstruction rate are similar in all our experiments. These go from 37% when 2,000 queries are made to 49% when 10,000 queries are made. This is likely because more queries lead to more intersections in the initial paths, helping to refine the candidate sets better. The token reconstruction rate with repetition is significantly higher than the other metrics. With 10,000 queries, the token reconstruction rate with repetition is over 70%. That is, for every query, the attacker can guess 70% of the characters correctly.

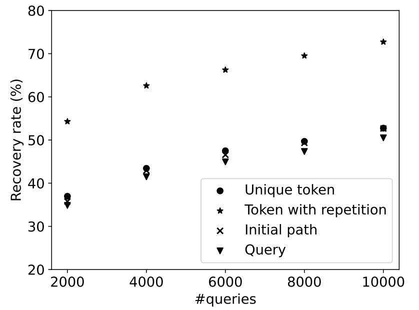
For the last set of experiments, we use 20,000 to 100,000 Wikipedia pages as the target dataset and 10,000 queries (with lengths between 3 to 11) to generate the leakage. The results are shown in Figure 2b. We do not see any significant trend in the metrics with respect to the number of strings used. This suggests that the number of strings is not a limiting factor for our attack and 20,000 strings are sufficient for our attack to perform well.

5.2 Experiments on Genome Dataset

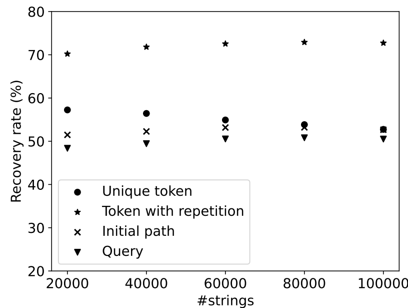
Experimental Data and Auxiliary Information. From the original NCBI genome dataset, we use half of the data to build the target dataset for our attack and reserve the other half of the data as the auxiliary data. The scale of the experiments in terms of the length of the texts in the target and auxiliary datasets on the genome dataset is comparable to that of the experiments with 100,000 Simple English Wikipedia pages.

Attack Parameters. The attack parameters are chosen in a manner similar to our attack experiments on Simple English Wikipedia.

[7] https://simple.wikipedia.org/wiki/Alan_Turing



(a) With respect to the number of queries.



(b) With respect to the number of strings.

Figure 2: Effectiveness of our attack on Simple English Wikipedia.

In particular, we repeat the same parameter-tuning experiments described above on the Genome dataset (with 10, 000 queries). The value of ε we use ranges from 3 to 7 and the value of t we use is either 2 or 3. The experimental results are shown in Table 3 in Appendix C.

We observe that the reconstruction space is significantly smaller for the Genome dataset as compared to the Simple English Wikipedia dataset. The hit rate for the Genome dataset is close to 100% for all of the tested parameters. We pick $\varepsilon = 5, t = 2$ as the parameters corresponding to the smallest reconstruction space.

Query Generation. We use the same process to generate the queries as before. The conditions for the queries to be considered valid are as follows: (1) the query must have length between 3 and 9, and (2) the query response volume must be larger than 100 (this is to avoid queries on erroneous genome sequences [62]).

Experiments. We present one set of experiments on the genome dataset to investigate how the attack scales with the number of queries. We generate between 2, 000 and 10, 000 queries and run our attack on the leakages resulting from the queries. Our results are reported in Figure 3. We are able to recover between 63% and 72% of the queries correctly and between 87% and 92% of the characters of the queries correctly. This shows that databases with a smaller alphabet size are significantly more vulnerable to our attack.

6 CONCLUSION

Our work initiates the study of leakage cryptanalysis of substring-SSE, focussing on the seminal scheme due to Chase and Shen from PoPETS'15 [14]. We proposed a query reconstruction attack against

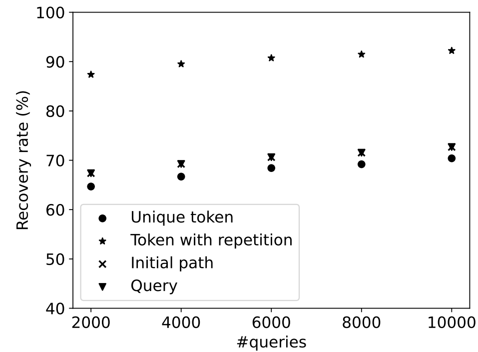


Figure 3: Effectiveness of our attack with respect to the number of queries on the NCBI genome dataset.

this scheme. We implemented our attack and experimentally validated its high query-recovery rate and practical efficiency over two real-world datasets – English Wikipedia and a genome dataset from NCBI. Our experiments show that our attack achieves high success rate with reasonable practical efficiency and that it scales to large datasets. For the English Wikipedia (100, 000 Wikipedia pages, 2, 000 queries), our attack recovers over 50% of the queries successfully, with 70% character recovery rate (64% and 80%, respectively, if we also include short queries of length 1 or 2). For the NCBI genome dataset (similar number of characters as 100, 000 Wikipedia pages, 2, 000 queries), we achieve over 60% query and character recovery rate.

An interesting direction of future research would be to develop query (and possibly even data) reconstruction attacks on other substring-SSE schemes [20, 36, 51]. This seems to require new insights, since each of the schemes has significantly different features and leakage. As has been done with SSE schemes for range queries [30–32, 44, 49], it would then be valuable to try to develop attacks that are *generic* in the sense of only requiring certain, limited types of leakage for their operation. This could lead to a better understanding of the privacy impact of different kinds of leakage in substring-SSE and inform the design of future schemes.

A number of recent works [17, 25, 35, 42, 57] have attempted to provide provably secure defenses against leakage cryptanalysis of SSE for keyword queries. Such an approach would also be valuable in the context of substring-SSE. We also leave it as a challenging open question to design either empirical or provably secure countermeasures against our query reconstruction attack on the scheme of Chase and Shen.

REFERENCES

[1] Bieganski, Riedl, Cartis, and Retzel. 1994. Generalized suffix trees for biological sequence data: applications and implementation. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, Vol. 5. 35–44. https://doi.org/10.1109/HICSS.1994.323593

[2] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2020. Revisiting Leakage Abuse Attacks. In *ISOC Network and Distributed System Security Symposium – NDSS 2020*. The Internet Society, San Diego, CA, USA.

[3] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-Preserving Symmetric Encryption. In *Advances in Cryptology – EUROCRYPT 2009 (Lecture Notes in Computer Science, Vol. 5479)*, Antoine Joux (Ed.). Springer, Heidelberg, Germany, Cologne, Germany, 224–241. https://doi.org/10.

1007/978-3-642-01001-9_13

[4] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *Advances in Cryptology – CRYPTO 2011 (Lecture Notes in Computer Science, Vol. 6841)*, Phillip Rogaway (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 578–595. https://doi.org/10.1007/978-3-642-22792-9_33

[5] Charlotte Bonte and Ilia Iliashenko. 2020. Homomorphic String Search with Constant Multiplicative Depth. In *ACM SIGSAC CCSW'20*, Yinqian Zhang and Radu Sion (Eds.). 105–117.

[6] Raphael Bost. 2016. $\Sigma o \phi o \varsigma$: Forward Secure Searchable Encryption. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1143–1154. https://doi.org/10.1145/2976749.2978303

[7] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *ACM CCS 2017: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 1465–1482. https://doi.org/10.1145/3133956.3133980

[8] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, Denver, CO, USA, 668–679. https://doi.org/10.1145/2810103.2813700

[9] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *ISOC Network and Distributed System Security Symposium – NDSS 2014*. The Internet Society, San Diego, CA, USA.

[10] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology – CRYPTO 2013, Part I (Lecture Notes in Computer Science, Vol. 8042)*, Ran Canetti and Juan A. Garay (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 353–373. https://doi.org/10.1007/978-3-642-40041-4_20

[11] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 1038–1055. https://doi.org/10.1145/3243734.3243833

[12] Yan-Cheng Chang and Michael Mitzenmacher. 2005. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS 05: 3rd International Conference on Applied Cryptography and Network Security (Lecture Notes in Computer Science, Vol. 3531)*, John Ioannidis, Angelos Keromytis, and Moti Yung (Eds.). Springer, Heidelberg, Germany, New York, NY, USA, 442–455. https://doi.org/10.1007/11496137_30

[13] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *Advances in Cryptology – ASIACRYPT 2010 (Lecture Notes in Computer Science, Vol. 6477)*, Masayuki Abe (Ed.). Springer, Heidelberg, Germany, Singapore, 577–594. https://doi.org/10.1007/978-3-642-17373-8_33

[14] Melissa Chase and Emily Shen. 2015. Substring-Searchable Symmetric Encryption. *Proceedings on Privacy Enhancing Technologies* 2015, 2 (April 2015), 263–281. https://doi.org/10.1515/popets-2015-0014

[15] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 2006: 13th Conference on Computer and Communications Security*, Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati (Eds.). ACM Press, Alexandria, Virginia, USA, 79–88. https://doi.org/10.1145/1180405.1180417

[16] Marc Damie, Florian Hahn, and Andreas Peter. 2021. A Highly Accurate Query-Recovery Attack against Searchable Encryption using Non-Indexed Documents. In *USENIX Security 2021: 30th USENIX Security Symposium*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 143–160.

[17] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage. In *USENIX Security 2020: 29th USENIX Security Symposium*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2433–2450.

[18] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos N. Garofalakis. 2016. Practical Private Range Search Revisited. In *ACM SIGMOD 2016*. 185–198.

[19] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos N. Garofalakis, and Charalampos Papamanthou. 2018. Practical Private Range Search in Depth. *ACM Trans. Database Syst.* 43, 1 (2018), 2:1–2:52.

[20] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS 2015: 20th European Symposium on Research in Computer Security, Part II (Lecture Notes in Computer Science, Vol. 9327)*, Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl (Eds.). Springer, Heidelberg, Germany, Vienna, Austria, 123–145. https://doi.org/10.1007/978-3-319-24177-7_7

[21] Francesca Falzon, Evangelia Anna Markatou, Akshima, David Cash, Adam Rivkin, Jesse Stern, and Roberto Tamassia. 2020. Full Database Reconstruction in Two Dimensions. In *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 443–460. https://doi.org/10.1145/3372297.3417275

[22] Paolo Ferragina and Giovanni Manzini. 2000. Opportunistic Data Structures with Applications. In *41st Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Redondo Beach, CA, USA, 390–398. https://doi.org/10.1109/SFCS.2000.892127

[23] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581.

[24] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *41st Annual ACM Symposium on Theory of Computing*, Michael Mitzenmacher (Ed.). ACM Press, Bethesda, MD, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[25] Marilyn George, Seny Kamara, and Tarik Moataz. 2021. Structured Encryption and Dynamic Leakage Suppression. In *Advances in Cryptology – EUROCRYPT 2021, Part III (Lecture Notes in Computer Science, Vol. 12698)*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer, Heidelberg, Germany, Zagreb, Croatia, 370–396. https://doi.org/10.1007/978-3-030-77883-5_13

[26] Eu-Jin Goh. 2003. Secure Indexes. Cryptology ePrint Archive, Report 2003/216. https://eprint.iacr.org/2003/216.

[27] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *19th Annual ACM Symposium on Theory of Computing*, Alfred Aho (Ed.). ACM Press, New York City, NY, USA, 182–194. https://doi.org/10.1145/28395.28416

[28] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

[29] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency Smoothing for Encrypted Data Stores. In *USENIX Security 2020: 29th USENIX Security Symposium*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2451–2468.

[30] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 315–331. https://doi.org/10.1145/3243734.3243864

[31] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2019. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1067–1083. https://doi.org/10.1109/SP.2019.00030

[32] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *ACM CCS 2019: 26th Conference on Computer and Communications Security*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, London, UK, 361–378. https://doi.org/10.1145/3319535.3363210

[33] Zichen Gui, Kenneth G. Paterson, and Sikhar Patranabis. 2023. Rethinking Searchable Symmetric Encryption. In *IEEE Symposium on Security and Privacy 2023*. IEEE, 1401–1418.

[34] Zichen Gui, Kenneth G. Paterson, Sikhar Patranabis, and Bogdan Warinschi. 2020. SWiSSSE: System-Wide Security for Searchable Symmetric Encryption. Cryptology ePrint Archive, Report 2020/1328. https://eprint.iacr.org/2020/1328.

[35] Zichen Gui, Kenneth G. Paterson, Sikhar Patranabis, and Bogdan Warinschi. 2024. SWiSSSE: System-Wide Security for Searchable Symmetric Encryption. *Proc. Priv. Enhancing Technol.* 2024, 1 (2024), 549–581.

[36] Florian Hahn, Nicolas Loza, and Florian Kerschbaum. 2018. Practical and Secure Substring Search. In *SIGMOD 2018*. ACM, 163–176.

[37] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *ISOC Network and Distributed System Security Symposium – NDSS 2012*. The Internet Society, San Diego, CA, USA.

[38] Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Outsourced symmetric private information retrieval. In *ACM CCS 2013: 20th Conference on Computer and Communications Security*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 875–888. https://doi.org/10.1145/2508859.2516730

[39] Charanjit S. Jutla and Sikhar Patranabis. 2022. Efficient Searchable Symmetric Encryption for Join Queries. In *Advances in Cryptology – ASIACRYPT 2022, Part III (Lecture Notes in Computer Science, Vol. 13793)*, Shweta Agrawal and Dongdai Lin (Eds.). Springer, Heidelberg, Germany, Taipei, Taiwan, 304–333. https://doi.org/10.1007/978-3-031-22969-5_11

[40] Seny Kamara and Tarik Moataz. 2017. Boolean Searchable Symmetric Encryption with Worst-Case Sub-linear Complexity. In *Advances in Cryptology – EUROCRYPT 2017, Part III (Lecture Notes in Computer Science, Vol. 10212)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.). Springer, Heidelberg, Germany, Paris, France, 94–124. https://doi.org/10.1007/978-3-319-56617-7_4

[41] Seny Kamara and Tarik Moataz. 2018. SQL on Structurally-Encrypted Databases. In *Advances in Cryptology – ASIACRYPT 2018, Part I (Lecture Notes in Computer Science, Vol. 11272)*, Thomas Peyrin and Steven Galbraith (Eds.). Springer, Heidelberg, Germany, Brisbane, Queensland, Australia, 149–180. https://doi.org/10.1007/978-3-030-03326-2_6
[42] Seny Kamara and Tarik Moataz. 2019. Computationally Volume-Hiding Structured Encryption. In *Advances in Cryptology – EUROCRYPT 2019, Part II (Lecture Notes in Computer Science, Vol. 11477)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, Germany, Darmstadt, Germany, 183–213. https://doi.org/10.1007/978-3-030-17656-3_7
[43] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *ACM CCS 2012: 19th Conference on Computer and Communications Security*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, Raleigh, NC, USA, 965–976. https://doi.org/10.1145/2382196.2382298
[44] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1329–1340. https://doi.org/10.1145/2976749.2978386
[45] Evgenios M. Kornaropoulos, Nathaniel Moyer, Charalampos Papamanthou, and Alexandros Psomas. 2022. Leakage Inversion: Towards Quantifying Privacy in Searchable Encryption. In *ACM CCS 2022: 29th Conference on Computer and Communications Security*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, Los Angeles, CA, USA, 1829–1842. https://doi.org/10.1145/3548606.3560593
[46] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2019. Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1033–1050. https://doi.org/10.1109/SP.2019.00015
[47] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2020. The State of the Uniform: Attacks on Encrypted Databases Beyond the Uniform Query Distribution. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1223–1240. https://doi.org/10.1109/SP40000.2020.00029
[48] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2021. Response-Hiding Encrypted Ranges: Revisiting Security via Parametrized Leakage-Abuse Attacks. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1502–1519. https://doi.org/10.1109/SP40001.2021.00044
[49] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 297–314. https://doi.org/10.1109/SP.2018.00002
[50] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shifeng Sun, Dongxi Liu, and Cong Zuo. 2018. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 745–762. https://doi.org/10.1145/3243734.3243753
[51] Iraklis Leontiadis and Ming Li. 2018. Storage Efficient Substring Searchable Symmetric Encryption. In *SCC@AsiaCCS 2018*. ACM, 3–13.
[52] Evangelia Anna Markatou, Francesca Falzon, Roberto Tamassia, and William Schor. 2021. Reconstructing with Less: Leakage Abuse Attacks in Two Dimensions. In *ACM CCS 2021: 28th Conference on Computer and Communications Security*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 2243–2261. https://doi.org/10.1145/3460120.3484552
[53] Jianting Ning, Xinyi Huang, Geong Sen Poh, Jiaming Yuan, Yingjiu Li, Jian Weng, and Robert H. Deng. 2021. LEAP: Leakage-Abuse Attack on Efficiently Deployable, Efficiently Searchable Encryption with Partially Known Dataset. In *ACM CCS 2021: 28th Conference on Computer and Communications Security*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 2307–2320. https://doi.org/10.1145/3460120.3484540
[54] Simon Oya and Florian Kerschbaum. 2021. Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption. In *USENIX Security 2021: 30th USENIX Security Symposium*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 127–142.
[55] Simon Oya and Florian Kerschbaum. 2022. IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization. In *USENIX Security 2022: 31st USENIX Security Symposium*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, Boston, MA, USA, 2407–2424.
[56] Sarvar Patel, Giuseppe Persiano, Joon Young Seo, and Kevin Yeo. 2021. Efficient Boolean Search over Encrypted Data with Reduced Leakage. In *Advances in Cryptology – ASIACRYPT 2021, Part III (Lecture Notes in Computer Science, Vol. 13092)*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer, Heidelberg, Germany, Singapore, 577–607. https://doi.org/10.1007/978-3-030-92078-4_20
[57] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing. In *ACM CCS 2019: 26th Conference on Computer and Communications Security*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, London, UK, 79–93. https://doi.org/10.1145/3319535.3354213
[58] Sikhar Patranabis and Debdeep Mukhopadhyay. 2021. Forward and Backward Private Conjunctive Searchable Symmetric Encryption. In *ISOC Network and Distributed System Security Symposium – NDSS 2021*. The Internet Society, Virtual.
[59] David Pouliot and Charles V. Wright. 2016. The Shadow Nemesis: Inference Attacks on Efficiently Deployable, Efficiently Searchable Encryption. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1341–1352. https://doi.org/10.1145/2976749.2978401
[60] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Oakland, CA, USA, 44–55. https://doi.org/10.1109/SECPRI.2000.848445
[61] Peter J. van Laarhoven. 1987. *Simulated annealing theory and applications*. Kluwer.
[62] Xin Victoria Wang, Natalie Blades, Jie Ding, Razvan Sultana, and Giovanni Parmigiani. 2012. Estimation of sequencing error rates in short reads. *BMC Bioinformatics* 13, 1 (30 Jul 2012), 185. https://doi.org/10.1186/1471-2105-13-185

A DERIVATION OF THE LIKELIHOOD FUNCTION

In this section, We give a concrete derivation of the likelihood function used in our attack. The likelihood function takes four inputs, namely:

- A guess guess : $\mathbb{N} \to \Sigma$. The allocation $\text{guess}(\text{tk}) = a$ means that our guess for token tk is letter a.
- Tokens $(\text{tk}_{i,1}, \ldots, \text{tk}_{i,m_i})_{i=1}^{l}$.
- A list of observed query response volumes $(\text{vol}_i)_{i=1}^{l}$, indicating that the i-th query has returned vol_i matching indices.
- An auxiliary distribution $\text{Aux} : \mathcal{P}(\Sigma) \to \mathbb{R}$. $\text{Aux}(s) = r$ means that we model the distribution of the query response volume for string s as a Poisson distribution with rate r.

We begin by writing down the probability of observing the query response volumes given the guess. The tokens and the auxiliary distribution will show up as constants in the expression.

$$\mathbf{Pr}[(\text{vol}_i)_{i=1}^{l} \mid \text{guess}] = \prod_{i=1}^{l} \mathbf{Pr}[\mathbf{Pois}(\text{Aux}((\text{guess}(\text{tk}_{i,1}), \ldots, \text{guess}(\text{tk}_{i,m_i})))) = \text{vol}_i]$$

In the expression, $s = (\text{guess}(\text{tk}_{i,1}), \ldots, \text{guess}(\text{tk}_{i,m_i}))$ is the guess we have for the i-th query (as a string). $\text{Aux}(s)$ gives the rate of the string in the auxiliary information and $\mathbf{Pr}[\mathbf{Pois}(\text{Aux}(s)) = \text{vol}_i]$ computes the probability of q_i having query response volume vol_i given the guess. Finally, the product in the end is due to our assumption on the independence of the distributions of the individual query response volumes.

Given the probability expression, we can use Bayes' theorem to turn the probability into a likelihood.

$$\begin{aligned} \mathbf{L}[\text{guess} \mid (\text{vol}_i)_{i=1}^{l}] &= \frac{\mathbf{Pr}[\text{guess}] \cdot \mathbf{Pr}[(\text{vol}_i)_{i=1}^{l} \mid \text{guess}]}{\mathbf{Pr}[(\text{vol}_i)_{i=1}^{l}]} \\ &\propto \mathbf{Pr}[\text{guess}] \cdot \mathbf{Pr}[(\text{vol}_i)_{i=1}^{l} \mid \text{guess}] \end{aligned}$$

We assume that all guesses are equally likely in our attack, so the likelihood function is proportional to $\mathbf{Pr}[(\text{vol}_i)_{i=1}^{l} \mid \text{guess}]$.

One caveat is that if $s = (\text{guess}(\text{tk}_{i,1}), \ldots, \text{guess}(\text{tk}_{i,m_i}))$ is not in Aux, then $\mathbf{Pr}[\mathbf{Pois}(\text{Aux}(s)) = \text{vol}_i]$ will be zero. This will make the whole likelihood function zero as well, due to its multiplicative nature. In practice, as the search space is sparse (meaning that only

ε	t	Reconstruction Space ($\log_{10}$)	Hit rate (%)
3	2	14268.36	98.84%
3	3	14330.88	98.88%
5	2	4517.15	99.93%
5	3	4617.95	99.93%
7	2	5245.26	99.99%
7	3	5351.36	99.99%

Table 3: The size of the reconstruction space and the hit rate for a selected few choices of ε and t on the NCBI genome dataset.

a small number of guesses yield non-zero likelihood), having zero in some of the multiplicative terms is unavoidable. On the other hand, we want to avoid these cases since the search algorithm we use depends on comparing the likelihoods of different guesses (zero is not helpful for this). For this reason, we use Laplace smoothing (with $\alpha = 1$) whenever $\mathsf{Aux}(s) = 0$. In addition, we use the log of the likelihood in our implementation to maintain accuracy in the otherwise very small numbers.

B DESCRIPTION OF THE DATASETS

Simple English Wikipedia. The Simple English Wikipedia is a collection of Wikipedia pages in Simple English words and grammar. There are over 220, 000 Simple English Wikipedia pages. We used the latest dump of the Simple English Wikipedia[8] in our experiments. We treat the text (after converting them to the lower case) in each Wikipedia page as a string. We randomly pick half of the strings as the target dataset and reserve the other half as the auxiliary dataset. For the target dataset, we build a generalised suffix tree and extract leakage from it by generating random queries. For the auxiliary dataset, we generate a generalised suffix tree and extract substring frequencies (only for the initial paths of the suffix tree) from it.

Genome Dataset. We obtained our genome dataset from the National Center for Biotechnology Information (NCBI).[9]. For a fair comparison between our attack on the Simple English Wikipedia and on a genome dataset, we decide to use a genome dataset that has text length similar to that of the Simple English Wikipedia. The dataset we used is the genome for Spialia galba.[10] We refer to the dataset as the genome dataset from here on. The genome dataset contains 283, 530 shotgun sequences (short fragments of genome) in total. We used half of the sequences as the target dataset and reserved the other half as the auxiliary dataset. We process the datasets in the same way as we have described for the Simple English Wikipedia.

C ATTACK PARAMETER SELECTION

ε	t	Reconstruction Space ($\log_{10}$)	Hit rate (%)
3	3	15039.46	80.13%
3	7	15640.57	82.37%
3	-	16654.44	84.41%
5	3	8680.20	87.95%
5	7	9890.36	89.25%
5	-	11158.75	89.79%
7	3	7757.57	90.90%
7	7	9287.07	91.13%
7	-	10787.98	91.32%

Table 2: The size of the reconstruction space and the hit rate for a selected few choices of ε and t on the Simple English Wikipedia. "-" in the t column means the trimming step is not executed.

[8] https://dumps.wikimedia.org/simplewiki/
[9] https://www.ncbi.nlm.nih.gov/
[10] https://www.ncbi.nlm.nih.gov/datasets/taxonomy/2705556/