File-Injection Attacks on Searchable Encryption, Based on Binomial Structures

Tjard Langhout, Huanhuan Chen, and Kaitai Liang

Delft University of Technology, Delft, The Netherlands
T.J.Langhout@student.tudelft.nl
{h.chen-2, kaitai.liang}@tudelft.nl

Abstract. One distinguishable feature of file-inject attacks on searchable encryption schemes is the 100% query recovery rate, i.e., confirming the corresponding keyword for each query. The main efficiency consideration of file-injection attacks is the number of injected files. In the work of Zhang *et al.* (USENIX 2016), $|\log_2 |K||$ injected files are required, each of which contains $|K|/2$ keywords for the keyword set K. Based on the construction of the uniform (s, n)-set, Wang et al. need fewer injected files when considering the threshold countermeasure. In this work, we propose a new attack that further reduces the number of injected files where Wang *et al.* need up to 38% more injections to achieve the same results. The attack is based on an increment (s, n)-set, which is also defined in this paper.

Keywords: Searchable encryption · File-injection attack · Binomial · Increment (s, n)-set

1 Introduction

Ensuring exclusive data access remains a paramount concern, often necessitating external cloud servers due to limited user storage capacity. To enable efficient data searches, these servers must implement search-over-plaintext methods for speed and efficacy.

Song *et al.* [18] were pioneers in proposing a cryptographic scheme tailored to address the challenge of searching encrypted data, particularly enabling controlled and concealed keyword searches. This general searchable encryption (SE) framework entails the storage of an index and database on the server. Each keyword within a file undergoes independent encryption, alongside the encryption of the file as a whole. Retrieval of files containing specific keywords involves the user generating a token by encrypting the desired keyword, which is then matched against all encrypted keywords stored on the server. Upon a match, the entire encrypted file is returned to the user for decryption. Since the introduction of this foundational scheme, numerous researchers have proposed diverse variants of SE schemes [2, 4, 5, 9, 12, 15, 19]. These schemes offer varying levels of file and keyword privacy, with the ORAM scheme emerging as the most secure, effectively concealing access pattern leakage [12]. However, schemes with minimal

leakage patterns tend to be computationally intensive and impractical. Alternatively, other proposed schemes, while computationally less burdensome, permit a marginally higher degree of leakage. Cash *et al.* [3] categorized these schemes into distinct leakage levels: L1, L2, L3, and L4, each revealing different degrees of information about keyword occurrences. Subsequent studies have demonstrated the potential exploitation of even minimal leakage to extract significant information from databases, emphasizing the critical role of prior knowledge in facilitating successful attacks [1, 8, 10, 13]. Recovery of keywords involves retrieving the keyword associated with the queried token, representing an encrypted keyword of a file.

Attacks on SE schemes may manifest as either passive or active. Passive attacks entail the observation of leakage patterns to construct keyword-query matches [6, 10, 14, 17]. These attacks refrain from interfering with protocols and leverage preexisting knowledge to execute their strategies. Passive attacks typically target weaker schemes exhibiting higher leakage levels (L2-L4) and often necessitate external or prior knowledge for execution. Conversely, active attacks involve servers injecting files into a user's database to glean insights. Injection attacks leverage either file access patterns or volume patterns [1, 3, 16, 20–22]. Active attacks, typically assuming L1 leakage or less, necessitate minimal prior knowledge, contrasting with the requirements of passive attacks. Successful recovery of keywords in active attacks is consistently achieved with 100% accuracy, with the performance metric being the number of injections required for a successful attack.

Cash *et al.* [3] were among the first to introduce an active attack wherein the server sends files to the client, subsequently encrypted and stored by the client. These attacks typically assume L2-L3 leakage, akin to passive attacks. Attackers construct files of their choosing and transmit them to users, compelling the application of the scheme to the received file, thereby enabling observation of ciphertext by the server. Zhang *et al.* [22] categorized such attacks as file-injection attacks and introduced the Binary-attack, premised on L1 leakage and injecting half of the keyword universe per injection, akin to binary search methodologies.

Countermeasures such as thresholds and padding are implemented to impede the success of attacks. Thresholds impose limits on the number of keywords a file can contain, while padding obscures actual results by introducing additional files alongside queried files. Wang *et al.* [20] proposed an alternative approach to injection attacks based on finite set theory, offering superior performance compared to previous methods. This approach, known as the FST-attack, necessitates fewer injections than the Binary-attack under certain conditions, leveraging so-called (s, n)-sets to enhance attack efficacy. Despite these advancements, the FST-attack's reliance on singular s values for all identified keywords represents a notable limitation.

Organization of the paper. The organization of the rest of the paper goes as follows. In Section 2, the description of the SSE scheme, file-injection, thresholds, and the latest state-of-the-art full file-injection attack on SSE schemes are given. In Section 3, our Binomial-attack is explained in detail, together with how

it can easily be applied under any threshold and dataset size, and the performances of our attack compared to previous file-injection attacks are visualised. In Section 4, we show the consequences of padding on our attack and compare these with the consequences on the FST-attack. In Section 5, a mitigation is proposed to perform better under a scheme that uses padding, with minimal trade-off. In Section 6 and 7 the results and untouched topics of the paper are debated. Finally, the paper is summarized in Section 8.

2 Preliminaries

2.1 Searchable Encryption

An *searchable encryption* (SE) scheme has three algorithms: encryption, search, and update (only for dynamic) algorithms.

The encryption algorithm takes as input a set of files $F = \{F_1, \cdots, F_n\}$ and a secret key from the data owner, and outputs the encrypted files. These ciphertexts are then stored on the cloud server. The search algorithm takes a secret key and a keyword k as input, and outputs a query(token) t, which allows the cloud server to search the files that contain the corresponding keyword k among the encrypted files. The data owner can then decrypt the returned documents from the server and identify all related files to the keyword k. The update algorithm only applies to the dynamic SE schemes, which outputs updated files, given a secret key and a set of files.

2.2 File Injection Attack

One of the goals of the attacker is called query recovery. The attack attempts to recover the underlying keywords to queries, which threatens query privacy and file privacy. We focus on file-injection attacks in this paper.

Instead of passive attacks, the attacker in file-injection attacks is active by sending to the data owner some proper documents, which are then encrypted by the latter and also stored on the cloud according to the SE schemes. As an example, one can inject files to a user by sending designed emails in the email system. The attacker then observes the returned files, especially its own injected files, corresponding to the queries through the search algorithm. According to the returned (previously injected) files, the attacker can achieve the goal of query recovery.

The first file-injection attack is proposed by Cash *et al.* [3] and further improved by Zhang *et al.* [22]. We show an example of the binary-search attack in Table 1 that injects $\log_2(|K|)$ files and achieves a 100% query recovery, where $|K|$ is the size of the keyword set. In this example, if returned files corresponding to a query t are F_1 and F_3, then we know its underlying keyword is k_2. Analogously, other keywords can also be matched according to different combinations of returned files.

In this work, our file-injection attack is based on the same assumption in [3, 22] that the attacker knows the file access pattern (i.e., knowing the returned files

Files	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8
F_1	1	1	1	1	0	0	0	0
F_2	1	0	1	0	1	0	1	0
F_3	1	1	0	0	1	1	0	0

Table 1: An example of the binary-attack file with a keyword universe of 8. Keywords are assigned into files: F_1, F_2, F_3, where 1 denotes the presence of the corresponding keyword and 0 indicates its absence.

according to queries) and also can identify the files on the cloud corresponding to its injected files. One distinguishable feature of file-inject attacks is the 100% query recovery rate, so we evaluate the efficiency of such attacks from the number of injected files.

Wang *et al.* [20] further improved the work [22] to deal with the countermeasures of a threshold of a maximal number of keywords in each file.

2.3 FST-Attack

In this section, we review the definition of a uniform (s, n)-set and how the FST-attack works [20], based on the uniform (s, n)-set. The method to construct a uniform (s, n)-set of a finite set is presented by Liu and Cao [11].

Definition 1 (Uniform (s, n)-set [20]). *Let a set $A = \{d_1, d_2, \cdots, d_m\}$ and the subsets $A_1, A_2, \cdots, A_n \subset A$ be called a uniform (s, n)-set of A ($m \geq \binom{n}{s-1}$) if the following three conditions are satisfied:*

- $|A_1| = |A_2| = \cdots = |A_n|$;
- *For any s subsets $A_{i_1}, \cdots, A_{i_s} \in \{A_1, \cdots, A_n\}$, there is $\bigcup_{j=1}^{s} A_{i_j} = A$;*
- *For any $s-1$ subsets $A_{i_1}, \cdots, A_{i_s-1} \in \{A_1, \cdots, A_n\}$ there is $\bigcup_{j=1}^{s-1} A_{i_j} = A \backslash \{d_i\}$.*

Where n denotes the number of injected files, $|A_i|$ the size of each injected file and m the keyword universe.

A uniform (s, n)-set for a finite set with size m has the following properties, when we choose $m = \binom{n}{s-1}$.

Lemma 1 ([20]). *Let $(A_1, A_2, \cdots, A_n)$ be a uniform (s, n)-set, then we have*

- *Size. The size of each file A_i is $|A_i| = \binom{n-1}{s-1}$ for $1 \leq i \leq n$.*
- *Intersection. Let $r = n - s + 1$, then the size of the intersection of arbitrary r files is only 1: $|\cap_{j=1}^{r} A_{i_j}| = 1$.*

Based on the uniform (s, n)-set, Wang *et al.* [20] presents a file-injection attack to SE. We assume the keyword set $\mathbb{K} = \{k_1, k_2, \cdots, k_m\}$.

Their basic attack is to first construct a uniform (s, n)-set $\{A_1, A_2, \cdots, A_n\}$ based on the technique presented by Liu and Cao [11] for the keyword set $\mathbb{K}$ such that $\binom{n}{s-1} \geq m$[1], and then generate a file set of size n: $\{D_1, D_2, \cdots, D_n\}$, where the file D_i contains the same keyword in the A_i for $1 \leq i \leq n$. Those files are then injected into the SE scheme, and the attack recovers the keyword corresponding to a token by the returned $n - s + 1$ files. The correctness of the basic attack is guaranteed by Lemma 1, i.e., there only exists one keyword in the intersection of $n - s + 1$ files.

When the threshold countermeasure is taken into consideration, that is the number of keywords in each file should be smaller than a threshold T, they proposed an advanced file-injection attack, aiming at obtaining a minimum n, the number of files that should be injected. Towards this goal, they choose the minimum n such that

$$\begin{cases} \binom{n-1}{s-1} \leq T \\ \binom{n}{s-1} \geq m. \end{cases} \tag{1}$$

Moreover, they present look-up tables to determine the optimal s and n corresponding to the threshold T and the number of keywords in different intervals.

Example 1. As an example of recovering 23 keywords $\{k_1, k_2, \cdots, k_{23}\}$ with threshold 7, we solve the Eq. (1) for $T = 7$ and $m = 23$ and then get a minimum $n = 8$ and $s = n - 1 = 7$. The corresponding injected files according to a uniform $(7, 8)$-set are shown in Table 2. Note that some parts in files $\{D_1, \cdots, D_8\}$ are left as blank since the number of keywords in these files has reached 23. Each keyword can be matched by $n - s + 1 = 2$ returned files. For example, if files D_1 and D_2 are returned after a query to a token t, we know the corresponding keyword to t is k_1.

This example also explains our major motivation: a single uniform (s, n)-set of the keyword set may not maximize the ability of a file injection attack, or in other words, the number of injected files n is not optimal. The reason is the number of keywords in injected files may be far from reaching the threshold.

3 A New File-injection Attack

In this chapter, we present our new file-injection attack to searchable encryption schemes. It is based on our new definition of a subset family of a finite set, called *increment* $[r, n]$*-set*. Our main technique is to construct an increment $[r, n]$-set of the keyword set with the help of a new construction method for the uniform (s, n)-sets. Compared to the uniform (s, n)-set used in [20], the

[1] It means the maximal number of keywords in the uniform (s, n)-set is greater than the keyword size m.

	$(7,8)$-set						
Files	Col_1	Col_2	Col_3	Col_4	Col_5	Col_6	Col_7
F_1	k_{22}	k_{23}					
F_2	k_{16}	k_{17}	k_{18}	k_{19}	k_{20}	k_{21}	
F_3	k_{11}	k_{12}	k_{13}	k_{14}	k_{15}	k_{21}	
F_4	k_7	k_8	k_9	k_{10}	k_{15}	k_{20}	
F_5	k_4	k_5	k_6	k_{10}	k_{14}	k_{19}	
F_6	k_2	k_3	k_6	k_9	k_{13}	k_{18}	
F_7	k_1	k_3	k_5	k_8	k_{12}	k_{17}	k_{23}
F_8	k_1	k_2	k_4	k_7	k_{11}	k_{16}	k_{22}

Table 2: An example of recovering 23 keywords with threshold T=7 by the uniform $(7,8)$-set.

increment $[r,n]$-set enables us to put more keywords in the injected files, thus significantly reducing the number of injected files.

3.1 New construction method of (s,n)-sets

To establish the integrity of our new (s,n)-set construct, we initially define a positional pattern as a distinct relative arrangement of files used to identify a keyword. Each positional pattern offers up to n variations before repeating itself. If all variants of a positional pattern are employed, they occupy at most $n \cdot r$ positions across r columns.

Consider the simplest positional pattern where files are consecutive. This pattern generates variants as illustrated in Table 3.

Table 3: An example of all the variants possible from a positional pattern (P.P.) for $n = 7$ and $r = 3$, and how they would be injected.

		Variants							**Injected structure**		
Files	**P.P.**	v_1	v_2	v_3	v_4	v_5	v_6	v_7	col_1	col_2	col_3
F_1	x	k_1					k_6	k_7	k_1	k_7	k_6
F_2	x	k_1	k_2					k_7	k_1	k_7	k_2
F_3	x	k_1	k_2	k_3					k_1	k_3	k_2
F_4			k_2	k_3	k_4				k_4	k_3	k_2
F_5				k_3	k_4	k_5			k_4	k_3	k_5
F_6					k_4	k_5	k_6		k_4	k_6	k_5
F_7						k_5	k_6	k_7	k_7	k_6	k_5

Given that each positional pattern occupies r columns (with exceptions noted for trivial edge cases, discussed later), patterns can be inserted into the files un-

til the threshold restricts usage to fewer than r columns. For the remaining columns, we consistently employ positional pattern (P.P. 1) from Table 3. Preceding columns in front of these columns can be filled with complete positional patterns, utilizing all variants of the pattern across r columns. See Table 4 for a visual clarification.

Table 4: An example of the construct of the injected files for $x_4r \leq T \leq x_5r$, $n = 7$ and $r = 3$. Where up till the last positional pattern (P.P.) random positional patterns can be used to make unique combinations and the last positional pattern consists out of consecutive files (P.P. 1). If $T = x_ir$, the threshold ends precisely in between positional patterns. If $x_ir < T < x_{i+1}r$, P.P. 1 should be used from x_ir onward. All columns in P.P. 1 can identify $\frac{n}{r}$ keywords.

		P.P.3			**P.P.4**			**P.P.5**			**P.P.1**			
Files		col_7	col_8	col_9	col_{10}	col_{11}	col_{12}	col_{13}	col_{14}	col_{15}	col_{16}	col_{17}	col_{18}	
F_1	$\cdots$	k_{15}	k_{18}	k_{21}	k_{22}	k_{25}	k_{28}	k_{29}	k_{31}	k_{35}	k_1	k_7	k_6	$\cdots$
F_2	$\cdots$	k_{15}	k_{16}	k_{19}	k_{22}	k_{23}	k_{26}	k_{29}	k_{30}	k_{32}	k_1	k_7	k_2	$\cdots$
F_3	$\cdots$	k_{16}	k_{17}	k_{20}	k_{23}	k_{24}	k_{27}	k_{30}	k_{31}	k_{33}	k_1	k_3	k_2	$\cdots$
F_4	$\cdots$	k_{17}	k_{18}	k_{21}	k_{24}	k_{25}	k_{28}	k_{31}	k_{32}	k_{34}	k_4	k_3	k_2	$\cdots$
F_5	$\cdots$	k_{15}	k_{18}	k_{19}	k_{22}	k_{25}	k_{26}	k_{32}	k_{33}	k_{35}	k_4	k_3	k_5	$\cdots$
F_6	$\cdots$	k_{16}	k_{19}	k_{20}	k_{23}	k_{26}	k_{27}	k_{29}	k_{33}	k_{34}	k_4	k_6	k_5	$\cdots$
F_7	$\cdots$	k_{17}	k_{20}	k_{21}	k_{24}	k_{27}	k_{28}	k_{30}	k_{34}	k_{35}	k_7	k_6	k_5	$\cdots$
$T =$		x_1r			x_2r			x_3r			x_4r	x_4r+1	x_4r+2	x_5r

Proof For our proof, we start by assuming $T = x_i \cdot r + 1$ and generalize from there. In the case of $T = x_i \cdot r + 1$, one column is left for the new positional pattern. In this column, there are three possible scenarios, where n_left stands for the number of free spots in the column available for keyword combinations:

1. $n\_left > r$: inject the variant starting from on the first available spot.
2. $n\_left < r$: inject the variant starting from on the first available spot and continue at the beginning of the next column.
3. $n\_left = r$: n/r is a an integer, inject the last variant to fully occupy the column.

The first scenario will ultimately progress to either the second or third scenario, both of which are illustrated in Table 5. In both scenarios, the column will eventually contain $\frac{n}{r}$ combinations. Let's now consider $T = x_i \cdot r + 2$, following from the previous scenarios which concluded with either scenario 2 or 3.

Table 5: Construction of the last $T \pmod r$ columns of an (s, n)-set under the two different possible scenarios, guaranteeing $\frac{n}{r}$ identifiable keywords per column.

(a) Second scenario

		P.P.1			
Files		col_{16}	col_{17}	col_{18}	
F_1	$\cdots$	k_1	k_7	k_6	$\cdots$
F_2	$\cdots$	k_1	k_7	k_2	$\cdots$
F_3	$\cdots$	k_1	k_3	k_2	$\cdots$
F_4	$\cdots$	k_4	k_3	k_2	$\cdots$
F_5	$\cdots$	k_4	k_3	k_5	$\cdots$
F_6	$\cdots$	k_4	k_6	k_5	$\cdots$
F_7	$\cdots$	k_7	k_6	k_5	$\cdots$
$T =$		x_4r	x_4r+2	x_4r+1	x_5r

(b) Third scenario

		P.P.1			
Files		col_{16}	col_{17}	col_{18}	
F_1	$\cdots$	k_1	k_5	k_6	$\cdots$
F_2	$\cdots$	k_1	k_2	k_6	$\cdots$
F_3	$\cdots$	k_1	k_2	k_3	$\cdots$
F_4	$\cdots$	k_4	k_2	k_3	$\cdots$
F_5	$\cdots$	k_4	k_5	k_3	$\cdots$
F_6	$\cdots$	k_4	k_5	k_6	$\cdots$
$T =$		x_4r	x_4r+2	x_4r+1	x_5r

- In the case of scenario 2: n and r are relatively prime. The variants will not repeat themselves for $n \cdot r$ spaces, equivalent to $\frac{n \cdot r}{n} = r$ columns. This means that variants will not repeat until $T = x_i \cdot r + r = x_{i+1} \cdot r$.
- In the case of scenario 3: $\frac{n}{r}$ is an integer, which makes n/r variants of the positional pattern. Repeating the same steps from the $T = x_i \cdot r + 1$ scenario, starting from a different unused position point, will not lead to repetition until all variants are utilized, totaling n variants. Together, these variants occupy $n \cdot r$ spaces, equivalent to $\frac{n \cdot r}{n} = r$ columns.

Repeating this for the next T will eventually lead to $T = x_i \cdot r + r$, after which the reasoning can be started from $T = x_{i+1} \cdot r + 1$ again.

This concludes that for any T, the described positional pattern P.P.1 can consistently identify $\frac{n}{r}$ keywords within a single column. By consistently using this positional pattern in the last $T \pmod r$ columns, we guarantee the identification of $\frac{n}{r}$ keywords per column.

edge case. if n and r are co-prime ($gcd(r, n) = 1$) all positional patterns have n variants and take r columns. If $gcd(r, n) = cp$, where $cp > 1$, there is at least 1 positional pattern that has less than n variants. namely $v = n/cp$ variants. If $cp/2$ is an integer, there is also a positional variant which has $v = n/(cp/2)$ variants, and so on. These take $\frac{v \cdot r}{n}$ columns per pattern. All other positional patterns have n variants and hence r columns. Note $\frac{v \cdot r}{n}$ still holds for $cp = 1$. The fact that there can be one or two positional patterns with less than r columns does not pose any trouble as P.P. 1 is always used as last positional pattern. Any positional pattern can be used for the construct of the files as long as the last pattern is the P.P. 1 pattern, to guarantee $\frac{n}{r}$ keywords per column.

3.2 Increment $[r, n]$-Set

The main idea of the increment $[r, n]$-set is to optimize the available space in the injected files, which are defined as follows.

Definition 2 (Increment $[r, n]$-set). *Let A be a set, then the subsets $A_1, A_2, \cdots, A_n \subset A$ are called an increment $[r, n]$-set of A if the following conditions are satisfied:*

- $|A_1| = |A_2| = \cdots = |A_{n-r+1}|$;
- *Elements in $A_1, A_2, \cdots, A_n$ are separated into r blocks such that the i-th $(1 \leq i \leq r)$ block of $A_1, A_2, \cdots, A_n$ forms a uniform $(n-i+1, n)$-set of the union set of the i-th block of $A_1, A_2, \cdots, A_n$.*

An (s, n)-set is here a single block and the increment $[r, n]$-set consists out of multiple (s, n)-sets (blocks), where the r is increased per block. Recall that for a uniform (s, n)-set, a keyword can be uniquely recovered by $r = n - s + 1$ returned files. Therefore, the keywords in the i-th block of an increment $[r, n]$-set are determined by $n-(n-i+1)+1 = i$ files, since the i-th block is a uniform $(n - i + 1, n)$-set by definition. That is, the keywords in the 1st block can be represented by 1 file, the keywords in the 2nd block by 2 files, and so on. This is what we call an *increment*.

We denote the i-th block of A_j by A_j^i for $1 \leq j \leq n$, then we get the following corollary which follows from Lemma 1.

Corollary 1. *If $(A_1, A_2, \cdots, A_n)$ is an increment $[r, n]$-set of A, then we have*

$$|A_j^i| = \binom{n-1}{n-i},$$

for $1 \leq i \leq r, 1 \leq j \leq n$.

Therefore, we know that the size of A_j is $|A_j| = \sum_{i=1}^{s} \binom{n-1}{n-i}$ for $1 \leq j \leq n$. The main idea of our basic file-injection attack is to construct an increment $[r, n]$-set, instead of complete independent (s, n)-sets spread over different chunks of files, like FST does. We are aiming at reducing the total number of injected files n to as few files as possible. Keywords are recovered according to the different combinations of returned files (details are present in Section [3.3]).

Example 2. We give an example of an increment $[r, n]$-set of the keyword set $\{k_1, k_2, \cdots, k_{23}\}$ with threshold seven for a comparison to the example in Table 2. We compute r and the minimum n such that

$$\begin{cases} \sum_{i=1}^{r} \binom{n-1}{n-i} \leq 7 \\ \sum_{i=1}^{r} \binom{n}{n-i} \geq 23, \end{cases} \tag{2}$$

	$(6,6)$-set	$(5,6)$-set					$(4,6)$-set
Files	Col_1	Col_2	Col_3	Col_4	Col_5	Col_6	Col_7
F_1	k_1	k_7	k_{10}	k_{13}	k_{15}	k_{19}	k_{22}
F_2	k_2	k_7	k_{11}	k_{14}	k_{16}	k_{20}	k_{22}
F_3	k_3	k_8	k_{11}	k_{13}	k_{17}	k_{21}	k_{22}
F_4	k_4	k_8	k_{12}	k_{14}	k_{18}	k_{19}	k_{23}
F_5	k_5	k_9	k_{12}	k_{15}	k_{17}	k_{20}	k_{23}
F_6	k_6	k_9	k_{10}	k_{16}	k_{18}	k_{21}	k_{23}

Table 6: An example of recovering 23 keywords with threshold T=7 by an increment $[3,6]$-set, which is divided into 3 blocks. Keywords in the 1st, 2nd, and 3rd block can be recovered by 1, 2, and 3 returned files, respectively.

and then we get $r = 3$ and $n = 6$. The increment $[3,6]$-set of the aimed keyword set is shown in Table 6. Compared to Example 1, it reduces the number of injected files from 8 to 6! Every space in these files is filled with keywords, while still controlling the total number of keywords within the threshold.

A trick to find the optimal n to Eq. 2 is to try r for values in $\{1, 2, \cdots\}$ in a row, and a general technique to construct an increment $[r, n]$-set for m keywords with threshold T is presented in Sect. 3.3.

3.3 Construction of Increment $[r, n]$-Set

In this section, we present a way to the construction of increment $[r, n]$-set of a finite set, which uses the method of constructing uniform (s, n)-set proposed in Sect. 3.1 as a subroutine.

Given as input the size of the keyword m and threshold of the number of keywords in a file T, we aim to construct an increment $[r, n]$-set of the keyword set with the minimum n such that (1) the size of each file should not be greater than the threshold T, and (2) the maximal number of keywords that those files can recover is at least m. To maximize the recovery ability under condition (1), our overall idea is to construct r uniform $(n - i + 1, n)$-sets by the technique shown in Sect. 3.1 for $1 \leq i \leq r$ and return the first T columns as the aimed set. Then by Lemma 1, we know the first $r-1$ blocks take $\sum_{i=1}^{r-1} \binom{n-1}{n-i}$ columns and can recover $\sum_{i=1}^{r-1} \binom{n}{n-i}$ keywords in total. The last block takes the rest $T - \sum_{i=1}^{r-1} \binom{n-1}{n-i}$ columns and allows to recover $\lfloor n/r \cdot [T - \sum_{i=1}^{r-1} \binom{n-1}{n-i}] \rfloor$ keywords. Then the condition (2) is equal to

$$\sum_{i=1}^{r-1} \binom{n}{n-i} + \left\lfloor \frac{n}{r} \cdot \left[T - \sum_{i=1}^{r-1} \binom{n-1}{n-i} \right] \right\rfloor \geq m. \tag{3}$$

We proceed in the discussion of r starting from 1 to T. For each r, we record all the possible n to the Inequality 3, with the minimum one as the optimal solution.

For simplicity of exposition, we denote $NK(r,n)$ as the left part of the above inequality. The whole process of constructing an increment $[r,n]$-set is present in Algorithm 1.

Algorithm 1 Construction of increment $[r,n]$-set

Input: Number of keywords m, threshold T
Output: An increment $[r,n]$-set of the keyword set $\{k_1, k_2, \cdots, k_m\}$

Initialize an empty candidate set: $candidate \leftarrow [\,]$
for $r = 1$ **to** T **do**
    Solve n from $NK(r,n) \geq T$ and denote the minimum n as n_0
    Append (r, n_0) to *candidate*
    $r = r + 1$
end for
Find (r, n_0) with the minimum n_0 and corresponding r from *candidate*
for $i = 1$ **to** r **do**
    Construct a uniform $(n_0 - i + 1, n_0)$-set of keywords with index from $\sum_{j=1}^{i-1} \binom{n_0-1}{n_0-j}$ to $\sum_{j=1}^{i} \binom{n_0-1}{n_0-j}$ by the technique proposed in Sect. 3.1
end for
Output the first T columns of the created files

Going back to Example 1, we compute the Inequality 3 to get $candidate = [(2,7),(3,6),(4,7)]$. Then we know the optimal increment $[r,n]$-set is $r = 3$, and $n = 6$.

3.4 Binomial-Attack

Given the keyword universe $\mathbb{K} = \{k_1, k_2, \cdots, k_m\}$ and the threshold T as the maximal number of keywords in each file, we present our file injection attack in Algorithm 2, which is based on the increment $[r,n]$-set of the $\mathbb{K}$.

Algorithm 2 Binomial-attack

Input: Keyword set $\mathbb{K} = \{k_1, k_2, \cdots, k_m\}$, threshold T, a query token t
Output: Keyword corresponding to the token t

Generate an increment $[r,n]$-set $A_1, A_2, \cdots, A_n$ of $\mathbb{K}$ with threshold T by Algorithm 1
for $j = 1$ **to** n **do**
    Let a file D_i contain the same keywords as A_i
end for
Inject files $\{F_1, F_2, \cdots, F_n\}$ into the SE scheme
return the corresponding keyword to t according to the returned i files$(1 \leq i \leq r)$

Now that the structure of the attack is understood, we can proceed to calculate the required number of injections to achieve the desired number of identifi-

able keywords. There are multiple formulas to calculate the required number of injections. The appropriate formula to utilize depends on at which $(n-r+1, n)$-set the threshold will limit the attack from injecting more combinations.

Deciding the Number of Injections. The attack always starts with $r = 1$ and progresses incrementally from there on forth. At some point within an $(n - r + 1, n)$-set, the threshold will limit the number of keywords it can inject. Refer to Table 6 for a visual representation.

By Eq. 3 we know the number of keywords an $(n - r + 1, n)$-set can utilize for a certain threshold, under a specific r. When the threshold is reached in the $(n - 1, n)$-set, where $r = 2$ the equation can be written in terms of n like the following:

$$F_2(K, T) = \frac{2K}{T + 1} \tag{4}$$

Similarly to the $(n - 2, n)$-set, where $r = 3$, the equation becomes:

$$F_3(K, T) = \frac{-(3 + 2T) + \sqrt{(3 + 2T)^2 + 24K}}{2} \tag{5}$$

The formulas $F_4(K, T)$ and beyond are only of relevance when the threshold is a significant portion of the number of keywords that need to be injected.

Deciding the Injection Formulas. The next step involves determining the appropriate utilization of each formula for different scenarios.

To determine the appropriate value for r in the increment $[r, n]$-set, we must assess whether the threshold allows for additional keywords in the files following a uniform $(n - r + 1, n)$-set. This evaluation must be conducted for each r, commencing at $r = 2$. By Lemma 1 we know the first two blocks utilize a total of n columns. Therefore if $T > n$, the $(n - 2, n)$-set can also be used. However, the value of n remains unknown at this stage. To address this uncertainty, we substitute $n = T$ into F_2. This yields the threshold at which both the $(n - r + 1, n)$-sets in the increment $[2, n]$-set become uniform and precisely meet the threshold. If the keyword universe exceeds this value, the attack will require more than T injections. Conversely, if the keyword universe falls below this value, fewer than T injections are required. Consequently, there will be residual space in the injected files for (at least) the $(n - 2, n)$-set. The minimum value of K to only be able to build up to an Increment $[2, n]$-set is outlined as follows:

$$Min_{F_2}(T) = \frac{1}{2}T^2 + \frac{1}{2}T \tag{6}$$

F_2 should be applied when the outcome of $Min_{F_2}(T) \leq K$. Alternatively, the formula of Min_{F_3} determines whether F_3 or F_4 should be utilized. Following the same procedures as before, we get:

$$Min_{F_3}(T) = \frac{2+T}{3} \cdot \frac{1+\sqrt{8T-7}}{2} + \frac{T-1}{3} \tag{7}$$

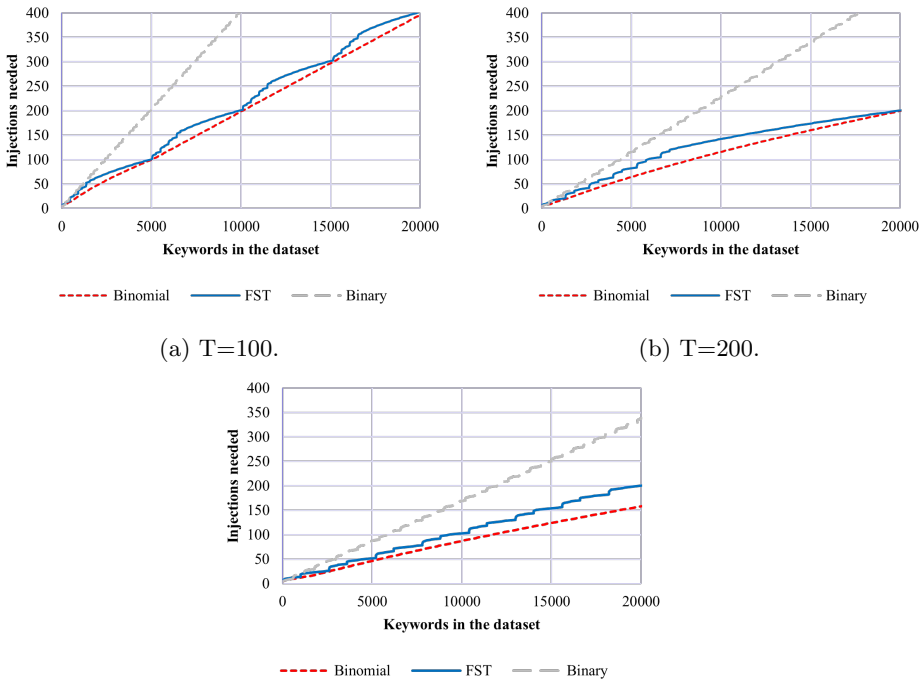
These formulas already hold an improvement over FST, since FST had a lookup table with overlapping values and no clear points to choose from. Using our previous example 1, we see $Min_{F_2}(7) > 23$ and $Min_{F_3}(7) < 23$. This means we need to use F_3, which results in $F_3(23, 7) = 6$ files.

3.5 Performance under Different Thresholds

The results consistently demonstrate the superiority of the Binomial-attack over the FST-attack across various thresholds and dataset sizes.

The Binomial-attack consistently outperforms the FST-attack with at least one injected file, regardless of the dataset size. With a threshold of 200, the most substantial disparity occurs in datasets ranging from 7 200 to 7 400 keywords, where the FST-attack requires 33 more injections compared to the Binomial-attack. This represents a 38% increase in injections needed by the FST-attack for equivalent results. In previous studies, the Enron dataset [7] served as a benchmark for attack performance. When applied to the Enron dataset, the FST-attack requires 83 files to cover the entire keyword universe, whereas the Binomial-attack accomplishes this with only 65 files. Thus, in this real dataset scenario, the FST-attack necessitates 28% more injections than the Binomial-attack.

To provide a comprehensive overview of these differences, Fig. 1 illustrates the comparative performance of the Binary-, FST-, and Binomial-attack across various thresholds, with datasets ranging up to 20 000 keywords.



(a) T=100.

(b) T=200.

(c) T=300.

Fig. 1: Performance of different injection attacks under different thresholds.

4 File-injection Attacks on SE Schemes with Keyword Padding

Keyword padding serves as a countermeasure within the SE scheme aimed at obscuring query results by returning more files than necessary. In addition to the files containing the queried keyword, the scheme also includes random files from the dataset in its response. In this chapter, we delve into the consequences of padding and compare the implications between the FST- and Binomial-attack methodologies. Previous studies, such as the FST- and Binary-attack, explored this topic assuming a file dataset of 30 109 files and a keyword universe of 5 050 keywords. The scheme adopts a threshold of 200, and on average, a query yields matches on 560 files, with an additional 60% of random files included (336 files). Section 4.1 delves into the quantitative effects of padding, while Section 4.2 presents a visual exploration of these effects.

4.1 Calculating the Effects

To assess the impact, three key steps are necessary. Firstly, we must determine the average number of additional injected files returned as a consequence of

their selection for padding. Subsequently, we can proceed to determine the average number of keyword combinations we can generate. These combinations represent distinct file arrangements utilized for the unique identification of a single keyword, collectively referred to as the candidate set for a query. Finally, the last step entails re-executing the attack on the candidate set to pinpoint the specific keyword utilized.

Injected Files from Padding. To calculate the average number of injected files chosen during padding, we can utilize the hypergeometric distribution function. Our population size is $30\,109 - 560 = 29\,549$, since the matched files for the query can not be chosen for the padding. The number of successes will be $F3(5\,050, 200) = 64.8 \approx 65$ files, minus the average injected file response, leaves $65 - 3 = 62$ successes. The sample size is 336. We can calculate the probabilities for all possible numbers of successes in the sample and then multiply each probability by the corresponding number of successes. The results are then summed to determine the average number of injected files (p) chosen in the padding:

$$p = \sum_{n=1}^{62} \frac{\binom{62}{n}\binom{29549-62}{336-n}}{\binom{29549}{336}} \tag{8}$$

Average Candidate Set Size. The average candidate set size is determined by three key factors associated with each $(n-r+1, n)$-set used to identify keywords. The first factor considers the number of possible combinations within the given $(n-r+1, n)$-set when $r+p$ injected files are returned. The second factor accounts for the ratio of combinations utilized in that $(n-r+1, n)$-set compared to its total possible combinations. The third factor represents the ratio of identifiable keywords in the $(n-r+1, n)$-set to the total number of identifiable keywords. Multiplying these three factors together yields the average candidate set size per $(n-r+1, n)$-set. Summing the results across all $(n-r+1, n)$-sets provides the overall average candidate set size:

$$\sum_{r=1}^{R} \binom{r+p}{r} \cdot \frac{|K_r|^2}{\binom{n}{r} \cdot |K_R|} \tag{9}$$

Number of Extra Injections Needed. A straightforward method to determine the number of extra injections required is to analyze on a per-query basis. By considering the average candidate set size per query, we can execute our attack specifically for that particular candidate set to recover the searched keyword. While this approach is not optimal, it suffices for comparison purposes with the FST-attack.

4.2 Visualising the Effects

This section will demonstrate the effects of padding on both FST and the Binomial-attack. While the Binomial-attack may not always appear significantly

better based solely on the average candidate set size per query, it's important to consider that FST is generally less efficient, requiring more injections to cover the same candidate set. Here, we present the results for a scheme with a threshold of 200. Results for different thresholds are available in Appendix A.

Targeting the Whole Dataset. In Fig. 2, we see the average sizes of candidate sets for different dataset sizes. The corresponding number of extra injections required for the candidate sets is illustrated in Fig. 3. While there is a small dataset size range where the Binomial-attack requires one more injection than the FST-attack, FST generally performs worse for all other dataset sizes.

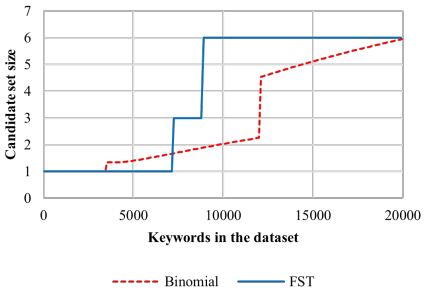


Fig. 2: Candidate set size per query, T=200.

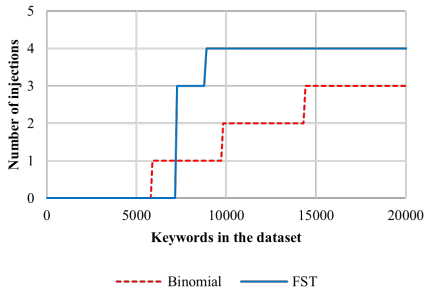


Fig. 3: Extra injection size per query, T=200.

Targeting a Subset of the Dataset. When targeting a subset of the keyword universe, fewer injections are required to cover the target set, benefiting both attacks. However, not every query relates to a keyword in the target set. When combined with padding, this may not pose an issue if we assume a consistent average number of injected files in the padding. For instance, if two injected files are returned and the average padding injection is also two, it suggests a search for a keyword not in the target set. However, if a return of two injected files could also indicate a search for a keyword occurring once or twice, all searches become candidate sets. While these candidate sets may not contain actual keywords from the target set, distinguishing beforehand is impossible. The only option is to re-perform the attack on the candidate set.

In this scenario, our attack performs notably worse. This is because the Binomial-attack initiates with an (n, n)-set. FST does not follow this approach, resulting in fewer potential combinations when all preceding $(n-r+1, n)$-sets are included in the candidate set. Figure 4 illustrates the number of extra injections required when searching for a keyword that is not in the target set.

5 Adopted Binomial-Attack

When the target set is a subset of the dataset, searches for keywords outside the target set result in additional candidate sets. To mitigate the size of these extra candidate sets, adjustments to the attack methodology are necessary. This chapter outlines the modifications required to minimize candidate size while maintaining effectiveness. Despite the trade-off, the attack consistently requires fewer initial injections than FST.

5.1 Removing the (n, n)-Set

In the Binomial-attack, the lowest value for r is always one. While this minimizes the space occupied in injected files, it also leads to greater overlap with keywords spread across multiple injected files. Conversely, higher values of r in the $(n - r + 1, n)$-sets for all keywords result in smaller candidate sets per query. To reduce the size of candidate sets, keywords should not be identified with only one injected file, meaning the attack starts from $(n-1, n)$ instead of (n, n). This frees up space that can be allocated to a different $(n - r + 1, n)$-set.

5.2 Results after the Mitigation

The number of identifiable keywords decreases by either $\frac{n}{2}$ or $\frac{2n}{3}$, depending on which $(n - r + 1, n)$-set the attack terminates due to the threshold. Refer to Table 7 for a visual representation of this transformation.

In Fig. 5, the difference in extra injections required between the FST- and adopted Binomial-attack is illustrated. FST consistently requires an equal or greater number of injections to recover candidate sets.

	$(5,6)$**-set**					$(4,6)$**-set**	
Files	Col_1	Col_2	Col_3	Col_4	Col_5	Col_6	Col_7
F_1	k_1	k_4	k_7	k_9	k_{13}	k_{16}	k_{19}
F_2	k_1	k_5	k_8	k_{10}	k_{14}	k_{16}	k_{18}
F_3	k_2	k_5	k_7	k_{11}	k_{15}	k_{16}	k_{18}
F_4	k_2	k_6	k_8	k_{12}	k_{13}	k_{17}	k_{18}
F_5	k_3	k_6	k_9	k_{11}	k_{14}	k_{17}	k_{19}
F_6	k_3	k_4	k_{10}	k_{12}	k_{15}	k_{17}	k_{19}

Table 7: Distribution of an Increment $[3, 6]$-set, without $(6, 6)$-set, T=7.

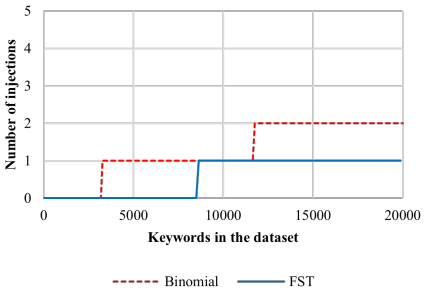


Fig. 4: Extra injection sizes per query that is not in the target set, T=200.

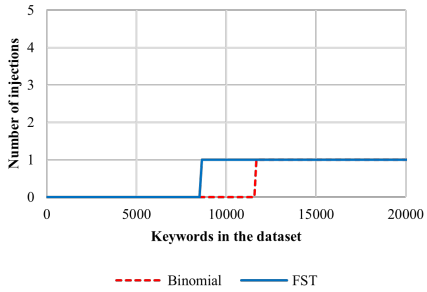


Fig. 5: Extra injection sizes per query that is not in the target set, T=200, for the adopted attack.

6 Discussion

In addition to padding, there exist other countermeasures aimed at increasing the difficulty of attacks. One such countermeasure involves the creation of clusters of keywords, as described in [10]. When a search query is initiated for one of the keywords within a cluster, all files containing keywords from the same cluster are returned. This approach not only obscures the specific keyword being searched for, but also introduces ambiguity regarding the association of injected files with specific keywords. Due to the potential for multiple combinations of keywords within the returned files, the attacker may be compelled to employ higher $(n - r + 1, n)$-sets, necessitating a greater number of injected files. It is important to note, however, that this countermeasure assumes a static keyword universe and may require modification to accommodate dynamic searchable encryption scenarios.

Despite its theoretical appeal, searchable encryption has yet to achieve widespread adoption in practical applications and can vary significantly in its configurations, including the implementation of countermeasures. Consequently, predicting the exact characteristics of a searchable encryption scheme in practice remains challenging. Nevertheless, there is value in speculating on the potential implications of different settings and attempting to assess the scheme's security under various conditions, even if these scenarios remain largely theoretical at present. This makes it harder to determine how big the safety issues of the schemes are.

7 Future work

The additional injections required to neutralize candidate sets are primarily utilized to compare the attack against FST. However, the method itself is far from

optimal. As presented in this paper and the FST paper, each keyword necessitates multiple additional injections. This approach may result in a greater number of injections than initially required for the attack. A more efficient strategy involves combining candidate sets and reusing earlier injections, thereby reducing the overall number of additional injections required. However, the optimal method for achieving this remains to be determined.

This attack is an active attack that makes no use of leakage apart from the returned injected files. In contrast, other attacks combine active and passive methods [21]. If Binary- or FST-attack methods are employed, they could be enhanced by incorporating the Binomial-attack. Revisiting these attacks may reveal potential improvements. We also note that further exploration into fields such as coding theory and combinatorics using our increment $[r, n]$-set could yield relevant connections and contributions and vice-versa.

8 Conclusion

The Binomial-attack represents a significant advancement over existing active attack methods. It maximizes the storage of keywords within a limited number of injected files by employing an Increment $[r, n]$-set to identify keywords. This approach iterates through all possible combinations of an $(n-r+1, n)$-set starting from $r = 1$, progressing with $r = r + 1$ until no additional space is available in the files. The adopted Binomial-attack starts at $r = 2$ to decrease the candidate set size for a query when the SE scheme uses padding as a countermeasure.

Our findings demonstrate that, regardless of the presence or absence of a threshold, the Binomial-attack consistently outperforms both the Binary- and FST-attack methods. However, when padding is introduced, there are specific threshold and dataset size combinations where FST requires fewer additional injections on average. It remains uncertain whether this advantage would persist with the implementation of a more efficient keyword recovery method.

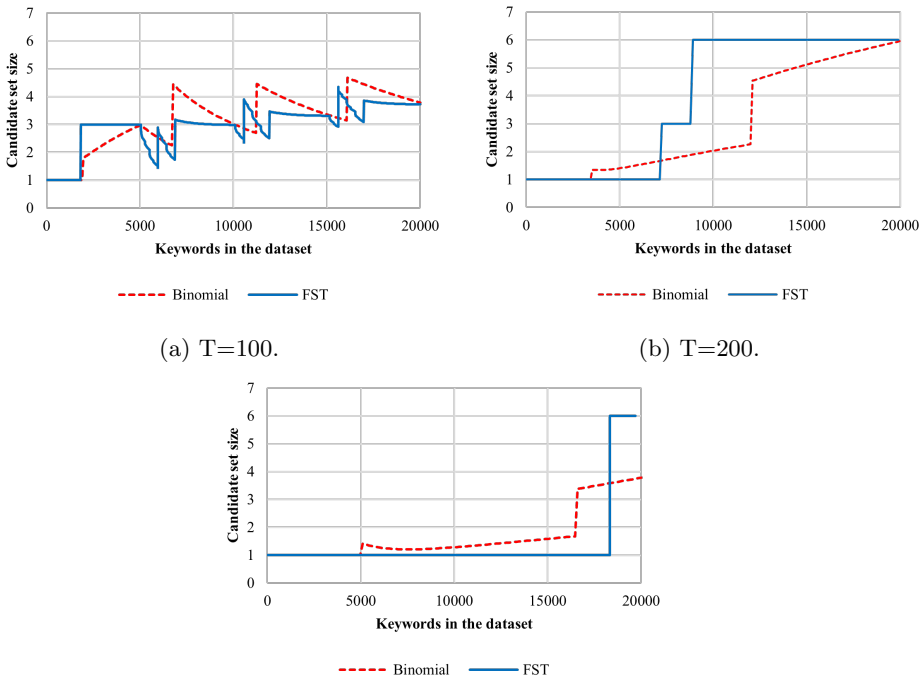
References

1. Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. In: NDSS 2020. The Internet Society (2020). https://doi.org/10.14722/ndss.2020.23103
2. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) CCS 2017. pp. 1465–1482. ACM (2017). https://doi.org/10.1145/3133956.3133980
3. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: CCS 2015. p. 668–679. Association for Computing Machinery (2015). https://doi.org/10.1145/2810103.2813700
4. Cash, D., Tessaro, S.: The locality of searchable symmetric encryption. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 351–368. Springer (2014). https://doi.org/10.1007/978-3-642-55220-5_20

5. Chamani, J.G., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) CCS 2018. pp. 1038–1055. ACM (2018). https://doi.org/10.1145/3243734.3243833
6. Damie, M., Hahn, F., Peter, A.: A highly accurate query-recovery attack against searchable encryption using non-indexed documents. In: Bailey, M.D., Greenstadt, R. (eds.) USENIX 2021. pp. 143–160. USENIX Association (2021)
7. Enron Corporation: Enron email dataset (2004), `http://www.cs.cmu.edu/~enron/`
8. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: NDSS 2012. The Internet Society (2012)
9. Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: Coron, J., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part III. LNCS, vol. 10212, pp. 94–124 (2017). https://doi.org/10.1007/978-3-319-56617-7_4
10. Liu, C., Zhu, L., Wang, M., an Tan, Y.: Search pattern leakage in searchable encryption: Attacks and new construction. Information Sciences **265**, 176–188 (2014). https://doi.org/10.1016/j.ins.2013.11.021
11. Liu, R., Cao, Z.F.: Two new methods of distributive management of cryptographic key. pp. 10–14. J. Commun., 8 (1987)
12. Naveed, M.: The fallacy of composition of oblivious ram and searchable encryption. IACR Cryptol. ePrint Arch. **2015**, 668 (2015), `https://api.semanticscholar.org/CorpusID:11042885`
13. Ning, J., Huang, X., Poh, G.S., Yuan, J., Li, Y., Weng, J., Deng, R.H.: LEAP: leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS 2021. pp. 2307–2320. ACM (2021). https://doi.org/10.1145/3460120.3484540
14. Oya, S., Kerschbaum, F.: Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In: Bailey, M.D., Greenstadt, R. (eds.) USENIX 2021. pp. 127–142. USENIX Association (2021)
15. Patel, S., Persiano, G., Yeo, K.: Symmetric searchable encryption with sharing and unsharing. In: López, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018, Part II. LNCS, vol. 11099, pp. 207–227. Springer (2018). https://doi.org/10.1007/978-3-319-98989-1_11
16. Poddar, R., Wang, S., Lu, J., Popa, R.A.: Practical volume-based attacks on encrypted databases. In: IEEE European Symposium on Security and Privacy, EuroS&P 2020. pp. 354–369. IEEE (2020). https://doi.org/10.1109/EUROSP48549.2020.00030
17. Pouliot, D., Wright, C.V.: The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) CCS 2016. pp. 1341–1352. ACM (2016). https://doi.org/10.1145/2976749.2978401
18. Song, D.X., Wagner, D.A., Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy. pp. 44–55. IEEE Computer Society (2000). https://doi.org/10.1109/SECPRI.2000.848445
19. Sun, S., Yuan, X., Liu, J.K., Steinfeld, R., Sakzad, A., Vo, V., Nepal, S.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) CCS 2018. pp. 763–780. ACM (2018). https://doi.org/10.1145/3243734.3243782

20. Wang, G., Cao, Z., Dong, X.: Improved file-injection attacks on searchable encryption using finite set theory. Comput. J. **64**(8), 1264–1276 (2021). https://doi.org/10.1093/COMJNL/BXAA161
21. Zhang, X., Wang, W., Xu, P., Yang, L.T., Liang, K.: High recovery with fewer injections: Practical binary volumetric injection attacks against dynamic searchable encryption. In: Calandrino, J.A., Troncoso, C. (eds.) USENIX Security 2023. pp. 5953–5970. USENIX Association (2023)
22. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: Holz, T., Savage, S. (eds.) USENIX Security 2016. pp. 707–720. USENIX Association (2016)

A Performance Comparison under Different Scenarios

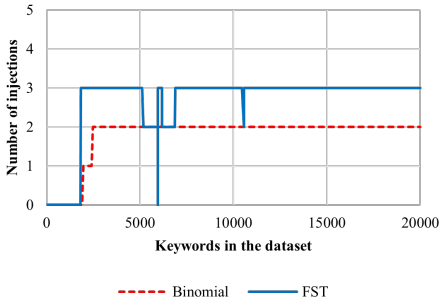


(a) T=100.

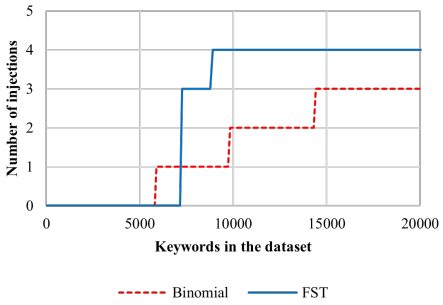
(b) T=200.

(c) T=300.

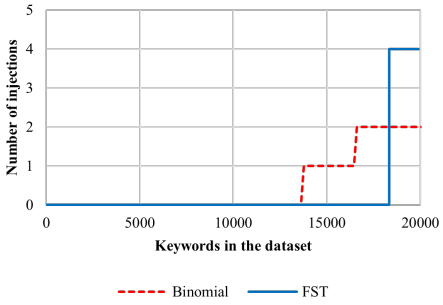
Fig. 6: Candidate set sizes per query when padding is applied, under different thresholds, where the target set is the full keyword universe, for the standard Binomial-attack.



(a) T=100.

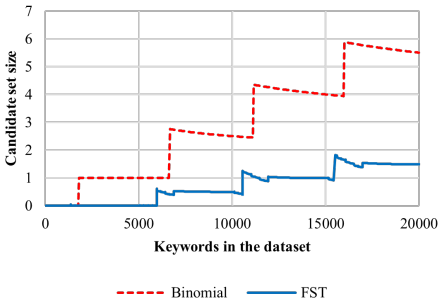


(b) T=200.

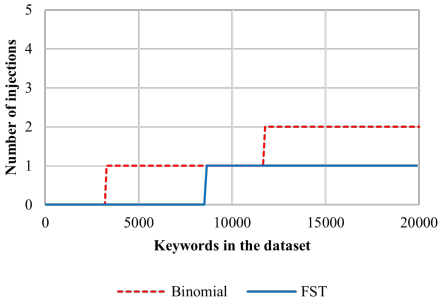


(c) T=300.

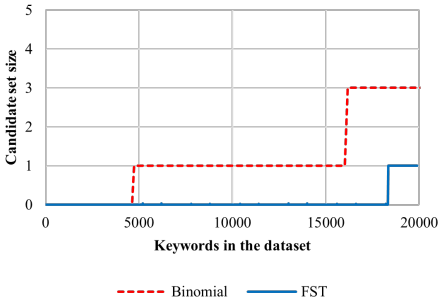
Fig. 7: Extra injection sizes per query when padding is applied, under different thresholds, where the target set is the full keyword universe, for the standard Binomial-attack.



(a) T=100.

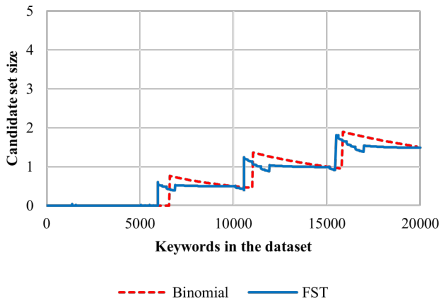


(b) T=200.

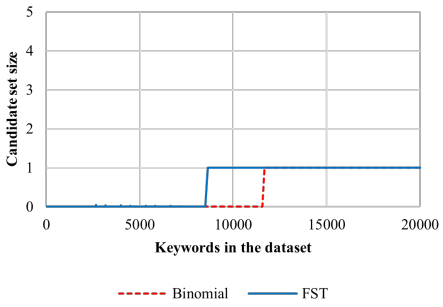


(c) T=300.

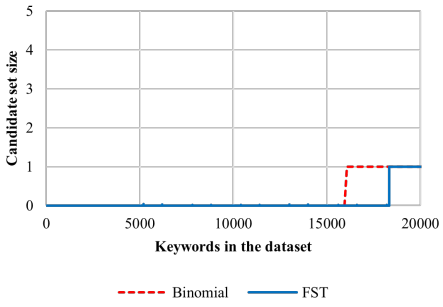
Fig. 8: Candidate set sizes per query that is not in the target set when padding is applied, under different thresholds, where the target set is a subset of the full keyword universe, for the standard Binomial-attack.



(a) T=100.

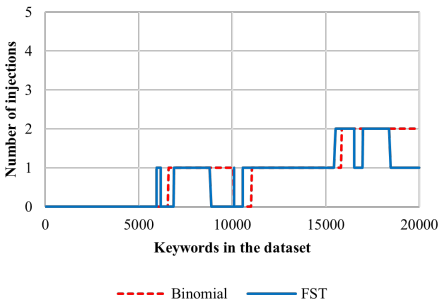


(b) T=200.

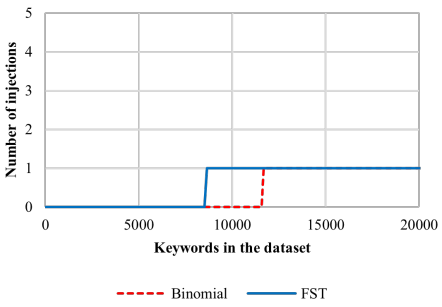


(c) T=300.

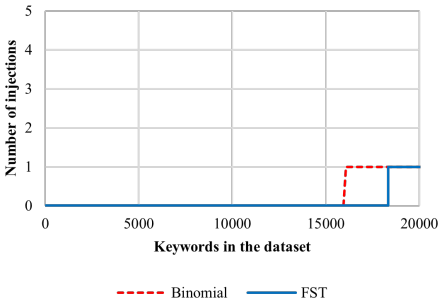
Fig. 9: Candidate set sizes per query that is not in the target set when padding is applied, under different thresholds, where the target set is a subset of the full keyword universe, for the adopted Binomial-attack.



(a) T=100.



(b) T=200.



(c) T=300.

Fig. 10: Extra injection sizes per query that is not in the target set when padding is applied, under different thresholds, where the target set is a subset of the full keyword universe, for the adopted Binomial-attack.