Parameterized Dictionary Matching with One Gap

B. Riva Shalom

Department of Software Engineering, Shenkar College, Ramat-Gan 52526, Israel.
rivash@shenkar.ac.il

Abstract. Dictionary Matching is a variant of the Pattern Matching problem where multiple patterns are simultaneously matched to a single text. In case the patterns contain sequences of don't care symbols, the problem is called Dictionary Matching with Gaps. Another famous variant of Pattern matching is the Parameterized Matching, where two equal-length strings are a parameterized match if there exists a bijection on the alphabets such that one string matches the other under the bijection. In this paper we suggest the problem of Parameterized Dictionary Matching with one Gap, stemming from cyber security, where the patterns are the malware sequences we want to detect in the text, and the necessity of a parameterized match is due to their encryption. We present two algorithms solving the Parameterized Dictionary Matching with one Gap. The first solves the problem for dictionaries with variable length gaps and has query time of $O(n(\beta_{\text{max}} - \alpha_{\text{min}}) \log^2 d + \text{occ})$, where $n$ is the size of the text, $d$ is the number of gapped patterns in the dictionary, $\beta_{\text{max}} - \alpha_{\text{min}}$ is the maximal size of gap and $\text{occ}$ is the number of the gapped patterns reported as output. The second solution considers dictionaries with a single set of gap boundaries and has query time of $O(n(\beta - \alpha) + \text{occ})$, where $n$ is the size of the text, $\beta - \alpha$ is the size of the gap and $\text{occ}$ is the number of the gapped patterns reported as output.

1 Introduction

Cyber security is a critical modern concern. It derives from cyber terrorist attacks, as well as economic dangers. Due to the importance of the problem, computer scientists develop various algorithms, dedicated to this struggle. Network intrusion detection systems perform protocol analysis, content searching and content matching, in order to detect harmful software. Such malware may appear on several packets, hence the need for gapped matching [24]. Having a list of gapped malware patterns yields the challenge of a dictionary matching with gaps.

In this paper we suggest an extension to the dictionary matching with one gap problem, where every pattern in the dictionary has a single gap, where the gapped malware is encrypted, in order to evade virus scanners. We consider the case in which the encryption used is substitution cipher, by which units of plain text are replaced with ciphertext, according to a fixed system, and consider a parameterized mapping as a strategy of encryption, thus define the Parameterized Dictionary Matching with One Gap (pDMOG) problem. We suggest an algorithm for dictionary with variable length gaps and another lower time complexity for dictionaries where all patterns have gaps with identical boundaries.

Since the pDMOG problem is a combination of the Dictionary Matching with one gap problem and Parameterized Matching problem, we define hereafter each of the problems separately then form the combined definition.

**Dictionary Matching with Gaps (DMOG)** Let a gapped pattern be of the form $P = lp\{\alpha, \beta\}rp$, where both the left subpattern $lp$ and the right subpattern $rp$
are strings over alphabet $\Sigma$, and $\{\alpha, \beta\}$ denotes a sequence of at least $\alpha$ and at most $\beta$ don’t cares symbols between the subpatterns, where a don’t care symbol can be matched to any text character from $\Sigma$. The formal definition follows.

**Definition 1.** The Dictionary Matching with One gap (DMOG) Problem:

- **Preprocess:** A dictionary $D$ of total size $|D|$ over alphabet $\Sigma$ consisting of $d$ gapped patterns each containing a single gap.
- **Query:** A text $T$ of length $n$ over alphabet $\Sigma$.
- **Output:** All locations $\ell$ in $T$, where any gapped pattern ends.

For example, let $D$ be the set of patterns $\{P_1 = a\, b\, a\, \{2, 4\}\, d\, d, P_2 = a\, b\, \{2, 4\}\, c\, d, P_3 = b\, a\, \{2, 4\}\, c\}$. Then, the text $T = c\, d\, a\, b\, a\, e\, b\, c\, d\, a\, c$ has occurrences of $P_2$ ending at location 10 with gap length of 4 and also with gap of length 2, and of $P_3$ ending at locations 9, with gap length of 3.

**Parameterized Matching** The Parameterized Matching problem is a well known problem in computer science, where two equal-length strings are a parameterized match if there exists a bijection on the alphabets such that one string matches the other under the bijection. Throughout the paper we denote a parameterized match by $p$-match. A formal definition follows.

**Definition 2.** Parameterized Matching Problem (PM):

- **Input:** A Text $T$ of length $n$ and a pattern $P$ of length $m$, both over alphabet $\Sigma \cup \Pi$, where $\Sigma \cap \Pi = \emptyset$.
- **Output:** All locations $\ell$ in $T$, where there exists a bijection $f : \Pi \to \Pi$ and the following hold:
  
  1. $\forall P[i] \in \Sigma$, $P[i] = T[\ell + i - 1]$.
  2. $\forall P[i] \in \Pi$, $f(P[i]) = T[\ell + i - 1]$.

For example, let $\Sigma = \{a, b\}$, $\Pi = \{x, y, z\}$ for text $T = x\, x\, y\, b\, z\, y\, y\, x\, b\, z\, x$ and pattern $P = z\, z\, x\, b\, z$ there are two p-matches ending at locations $\{4, 10\}$. The former implies mapping function $f(z) = x$, $f(x) = y$ while the latter implies mapping function $f(z) = y$, $f(x) = x$.

**Parameterized Dictionary Matching with One Gap (pDMOG)** The pDMOG problem is a combination of the above problems. Note, that according the motivation of the problem, we consider malicious code to appear on two packets, thus each part of the gapped pattern $lp_i, rp_i$, does not relate to the other part of the pattern, hence, they can be matched using different matching functions. The formal definition follows.

**Definition 3.** The Parametrized Dictionary Matching with One gap (pDMOG) Problem:

- **Preprocess:** A dictionary $D$ consisting of $d$ gapped patterns $\{P_i\}$ over alphabet $\Sigma \cup \Pi$, where $\Sigma \cap \Pi = \emptyset$ where every $P_i$ is of the form $lp_i(\alpha_i, \beta_i)rp_i$ and $\alpha_i, \beta_i$ are $P_i$’s gap boundaries.
- **Query:** A text $T$ of length $n$ over alphabet $\Sigma \cup \Pi$, $\Sigma \cap \Pi = \emptyset$.
- **Output:** All locations $\ell$ in $T$, where there exists a bijection $f : \Pi \to \Pi$ and all the following hold for any $P_i$ and a gap length $g \in [\alpha_i, \beta_i]$:
  
  1. $\forall lp_i[j] \in \Sigma$, $lp_i[j] = T[\ell - |lp_i| - j]$.
  2. $\forall rp_i[j] \in \Pi$, $f(lp_i[j]) = T[\ell + g + j]$.
  3. $\forall rp_i[j] \in \Pi$, $f(rp_i[j]) = T[\ell + g + j]$. 
  4. $\forall rp_i[j] \in \Pi$, $f(rp_i[j]) = T[\ell + g + j]$. 

For example, let $\Sigma = \{a, b\}, \Pi = \{q, u, v, w, z\}$ for text $T = a\ u\ v\ b\ b\ a\ z\ w\ w\ z$ and $D = \{P_1 = z\ x\ b\ z\{2, 4\} u\ u\ q, P_2 = u\ b\ q\{1, 4\} a\ u\ v\}$ we have two p-matches ending at locations $\{11, 9\}$. The former implies p-matching $P_1$ using mapping function $f(z) = u, f(x) = u$ for $lp_1$, a gap of length 3 and a mapping function $f(u) = w, f(q) = z$ for $rp_1$. The latter implies p-matching $P_2$ using mapping function $f(u) = v, f(q) = u$ for $lp_2$, a single character gap and a mapping function $f(u) = z, f(v) = w$ for $rp_2$.

We consider the alphabet to be of fixed size. If it is of variable size, a factor of $\log\sigma$ is to be multiplied to $n$ in the query time of both solutions.

The paper is organized as follows. Section 2 scans previous work. Section 3 suggests the framework of the algorithm and some notations. The first part of the algorithm appears on Section 4 and the second part of the algorithm appears on Section 5. Section 6 concludes the paper and poses some open problems.

## 2 Previous Work

Dictionary matching has been amply researched (see e.g. [2,3,4,5,7,15]). When the patterns are gapped, and we consider the problem of Dictionary Matching with Gaps, there are several algorithms solving the problem, yet their definitions of the problem are not identical.

Rahman et al. [28] suggest an algorithm using AC automaton, and suffix arrays built over the text. Bille et al. [14], [13] improved time complexity, by using sorted lists of disjoint intervals, yet both solutions includes a factor of $socc$ which is the total number of occurrences of the subpatterns in the text which can be very large. Kucherov and Rusinowitch [25] and Zhang et al. [29] solved the problem of matching a set of patterns with variable length of don’t cares. Yet, they report a leftmost occurrence of a pattern if there exists one, while we are interested in all occurrences of the patterns in the text. Haapasalo et al. [20] gave an on-line algorithm for the general problem, yet, they report at most one occurrence for each pattern at each text position.

Amir et al. [9] solved the DMOG problem for a single set of gap boundaries, reporting all appearances of all gapped patterns. They suggest an algorithm using range queries and an additional algorithm using a look-up table. The query time of their second algorithm is $O(|T|\beta - \alpha + occ)$ and space of $O(d^2 + |D|)$, where $d$ is the number of gapped patterns in dictionary $D$ and $occ$ is the number of patterns reported. Hon et al. [21] presented a similar solution, for dictionaries with variable length gaps, improving the space complexity to a linear space and requiring query time of $O(|T|\gamma \log \lambda \log d + occ)$, where $\gamma$ denotes the number of distinct gap lengths and $\lambda$ denotes the number of distinct lower and upper bounds of gap lengths.

Amir et al. [8] also considered the online version of the DMOG problem, where the text arrives online, a character at a time, and the requirement is to report all gapped patterns that are suffixes of the text that has arrived so far, before the next character arrives. In [10] Amir et al. considered the recognition version of the online DMOG problem, where each gapped pattern is reported at most once, during the entire online text scan.

Regarding Parameterized Matching, the problem was initially defined as a tool for software maintenance, motivated by the observation that programmers introduce duplicate code into large software systems when they add new features or fix bugs, thus
slightly modify the duplicated sections. The problem has many application in various fields, as detailed in [27], such as Image processing, where parameterized matching can help searching an icon on the screen, or improving ergonomy of databases of URLs. As a consequence, extensive work has been done on the problem and its various variants, some of which Lewenstein [26] and Mendivelso and Pinzon [27] scan.

Among the parameterized matching extensions are, the work of Amir et al. [6] suggesting a parameterized version of KMP, Baker works [12], [11] regarding the maximal p-matches over a threshold length and a p-suffix tree, the parameterized fixed and dynamic dictionary problems presented by Idury and Sahueffer [22], and improved by Ganguly et al. [19], the efficient parameterized text indexing, shown by Ferragina and Grossi [17], p-suffix arrays presented by Deguchi et al. [16], the Parameterized version of the LCS problem by Keller et al. [23] and many more.

3 Parameterized Dictionary Matching with One Gap - Framework

Throughout the paper we use the following notations. Let \( D = \{P_1, \ldots, P_d\} \) be the dictionary, where every \( P_i \) is a gapped pattern of the form \( l_{P_i}(\alpha_i, \beta_i) \, r_{P_i} \). In case the dictionary has a single set of gap boundaries \( \{\alpha, \beta\} \), then \( \forall 1 \leq i \leq d, \ \alpha_i = \alpha \) and \( \beta_i = \beta \). We call \( l_{P_i} \) the left subpattern of \( P_i \), and call \( r_{P_i} \) the right subpattern of \( P_i \). We divide all subpatterns of the dictionary into two sets \( \text{Left} =_{1 \leq i \leq d} \{l_{P_i}\} \) where \( d_{\text{Left}} = |\text{Left}| \leq d \) and \( \text{Right} =_{1 \leq i \leq d} \{r_{P_i}\} \) where \( d_{\text{Right}} = |\text{Right}| \leq d \).

The solution for the DMOG problem, suggested in this paper, follows the frameworks of [9] and their improvement in [21]. Their algorithms consist of two parts: The first part is detecting separately all the left subpatterns and all the right subpatterns of the dictionary in the text. The second part is processing the subpattern occurrences, in order to efficiently report all gapped patterns \( P_i \) where both their subpatterns appear with a \( g \) sized gap between them, where \( \alpha_i \leq g \leq \beta_i \).

For the first step they practiced the observation that matching two parts of a pattern \( P_i \) to a text, can be done by matching the reverse of the left subpatterns in \( \text{Left} \) to the reverse of \( T[1, \ldots, \ell] \) for all \( \ell \)s and matching the right subpatterns in \( \text{Right} \) to \( T[\ell + g + 1 \ldots n] \), where \( g \) is the size of the gap between the subpatterns occurrences. To this aim they constructed a generalized suffix tree of all the reverse of the \( \text{Left} \) subpatterns, and a generalized suffix tree of all the \( \text{Right} \) subpatterns.

For the second step, given a match of the reverse of some \( l_{P_i} \) to \( T[\ell \ldots 1] \) and a match of some \( r_{P_j} \) to \( T[\ell + g + 1 \ldots n] \), it is necessary to conclude which gapped patterns occurred, thus ought to be reported. Note, that several gapped patterns can be reported, such that their left subpattern is a suffix of \( l_{P_i} \) and their right subpattern is a prefix of \( r_{P_j} \).

Hon et al. [21] suggested using range queries by rectangular stabbing for the second step of the algorithm. [9] suggested an additional technique, when all patterns share the same gap boundaries, in case query time is required to be \( O(1 + \text{occ}) \) time per text location and gap length, where they use a look up table built in the preprocess stage.

The parameterized matching does not require exact matches between the \( \Pi \) characters, but rather to capture the characters order in the pattern. For this reason
Baker [11] defined a p-string over a string \( S = s_1, s_2 \cdots \) using the \( \text{prev} \) function, where \( \text{prev}(s_i) = s_i \) in case \( s_i \in \Sigma \), but for \( s_i \in \Pi \), \( \text{prev}(s_i) = 0 \) if \( s_i \) is the leftmost position in \( S \) of this character, and \( \text{prev}(s_i) = i - k \) if \( k \) is the previous position to the left at which the character \( s_i \) occurs. For example, let \( \Sigma = \{a, b\} \), \( \Pi = \{u, v\} \) and \( S = a \ b \ u \ v \ a \ b \ u \ v \ u \), then \( \text{prev}(S) = a \ b \ 0 \ 0 \ a \ b \ 4 \ 4 \ 2 \). The string obtained by \( \text{prev}(S) \) is called the p-string of \( S \).

**Lemma 4.** [11] Strings \( S_1, S_2 \) have \( \text{prev}(S_1) = \text{prev}(S_2) \) iff they are p-matched.

A direct result of this lemma is that using p-strings enables applying parameterize matching to various pattern matching techniques, with certain modifications required due to the behaviour of the \( \text{prev} \) function, some of which were referred to in Section [2].

This paper follows the frameworks of [9], [21], yet adapts them to using parameterized matching while solving the DMOG problem. The algorithms suggested are described in the following sections, according to the parts of the solution.

### 4 p-Matching of Subpatterns

As mentioned in the previous section, both [9] and [21] construct two generalized suffix trees, one over the Right subpatterns and the other over the reverse of the the Left subpatterns, which we call Left\(^R\). They do not traverse the text query \( T \) using these suffix trees but rather require to attain the longest match of every suffix of \( T \) with the Right suffix tree and the longest match of every suffix of \( T^R \) and the Left\(^R\) suffix tree. They do it in \( O(n) \) time using Amir et. al. [5] technique of inserting all suffixes of \( T \) or \( T^R \) to the corresponding suffix trees. By inserting each suffix of \( T \) to the subpatterns generalized suffix tree, the needed information is gathered in linear time.

When considering the parameterized matching case, the parameterized suffix tree is considered as the mechanism of locating the \( \text{prev} \) function of the subpatterns in \( \text{prev}(T) \). Baker [11] showed the construction of a parameterized suffix tree. She used dynamic trees and lowest common ancestor queries to achieve the following results.

**Lemma 5.** [11] Given finite disjoint alphabets \( \Sigma, \Pi \), a p-suffix tree can be built for a p-string \( S \) in time \( O(|S| \log |S|) \) and linear space in the \( |S| \). Given a p-string text query \( T \), all p-matches of \( S \) in \( T \) can be reported in time \( O(|T|) \) for fixed alphabets and in time \( O(|T| \log (\min\{|S|, \sigma\})) \), where \( \sigma = |\Sigma| + |\Pi| \) for variables alphabets.

Other works, such as [19] considered an efficient construction and space consumption of a p-suffix tree, yet they did not reduce the construction time from \( O(|S| \log |S|) \). The scheme we follow requires construction of p-suffix trees both in the preprocess and during query execution, where we insert suffixes of \( \text{prev}(T) \) to a generalized p-suffix tree, thus, the first part of answering a query requires \( O(n \log n) \) time. In order to decrease the query time, we consider another technique for dictionary matching for the first step of the algorithm, which is using the Aho-Corasick automaton [2].

Idury and Schaffer [22] constructed a modified Aho-Corasick automaton (AC) [2] suitable for p-strings. Their construction algorithm is similar to that of the original AC construction, yet important modifications were made to the goto and fail links of the automaton, adapting it to work with p-strings. Their p-AC automaton occupies \( O(m \log m) = O(|D| \log |D|) \) bits, where \( m \) is the number of states in the automaton. They report all p-matches of patterns from dictionary \( D \) in text \( T \) in \( O(|T| \log \sigma + \text{occ}) \) time, where \( \text{occ} \) are the number of reported occurrences. Note that in case we report
only the longest pattern located for each text location, the query is answered in $O(|T| \log \sigma)$ as the occ element is added since we require reporting all appearances of subpatterns that are suffixes of the longest subpattern recognized. Ganguly et. al. [18] suggested a space efficient data structure for the parameterized dictionary matching, improving the p-AC automaton of [22] by using sparsification technique. Their index requires $O(|D| \log \sigma + d \log |D|)$ bits and the report of all p-matches in text $T$ requires $O(|T| (\log \sigma + \log_d |D|) + occ)$. Due to our motivation in cyber security we use the data structure of Idury and Schaffer [22], guaranteeing a faster query time.

We calculate in linear time the p-string, prev(lp$_i$) for every $lp_i \in Left$ and construct a p-AC automaton upon them, named $LpAC$. In addition we calculate prev(T), thus by scanning prev(T) using $LpAC$ we can locate all left subpatterns p-matching the text $T[1..\ell]$, where the p-match ends at location $T[\ell]$. For the Right subpatterns, we need to locate all occurrences of prev(rp$_j$) starting at location $\ell$ in prev(T), hence, we need to scan the reverse of $T$ and look for occurrences of the prev(rp$_j^R$), therefore for every $rp_j \in Right$ we calculate in linear time the p-string of the reversed subpattern, prev(rp$_j^R$) and construct a p-AC automaton upon them, named $RpAC$. In addition, the prev function of the reverse of the text $prev(T^R)$ is calculated.

Note, that even in case the alphabets are not fixed, calculating the prev function of a string $S$ requires $O(|S|)$ time by using perfect hash tables for the position of the latest occurrence of a character in $S$. Each automaton consists of states, representing the p-strings of prefixes of the dictionary subpatterns. We consider the $p-label$ of a state to be the p-string of the sequence the state represents. A state p-labeled by a p-string of a subpattern from the dictionary is called an accepting state. Every state in the p-automata is numbered as will be described in the next section.

We scan prev(T) using the $LpAC$ automaton and for every location $\ell$ in prev(T), reached by the automaton, we save at array Locc[$\ell$] the number of the current state in $LpAC$. Similarly, we scan prev(T$^R$) using the $RpAC$ and save in Rocc[$\ell$] the number of the current state reached by the automaton at prev(T$^R$[$\ell$]).

**Lemma 6.** Performing the search with $LpAC$, $RpAC$ yields for each text location $\ell$, a state representing the longest prefix of some prev(lp$_i$), p-matching the suffix of prev(T[1..\ell]) and a state representing the longest prefix of some prev(rp$_j$) p-matching the prefix of prev(T[\ell..|T|]), in linear time in the length of the text, for fixed alphabets, and with $O(|D| \log |D| + n)$ space requirements, where $|D|$ is the size of the dictionary and $n$ is the size of the text.

**Proof.** Scanning prev(T) of the query text $T$ with both of the p-automata requires $O(n)$, as [22] proves that scanning a p-text with a p-automaton requires linear time in the size of the text, for fixed alphabets. A single scan is sufficient using each p-automaton as the parameterized fail links, pfail allow continuation of search from the point of a mismatch between the prev(T) and the prev of the current matched subpattern [22]. The p-automaton saves at every step of the scan the current state p-matching the current prev(T) character, thus the longest prefix of a p-subpattern ending at the current text location. Using the reverse of $T$ and the reverse of rp$_j$ sub-patterns we get that p-matching the longest prev(rp$_j^R$) at the suffix of prev(T[n..\ell]) equals the p-matching of the longest rp$_j$ starting at prev(T[\ell..|T|]).

Regarding space, the p-automaton is built over dictionary of size $|D|$, thus requires $O(|D| \log |D|)$ space, as proved in [22]. In addition we save the Locc, Rocc arrays maintaining a pointer to a single state, for every text location. □
5 Results Calculation

Given the output of the p-automata scans at arrays \( \text{Locc} \), \( \text{Rocc} \), the second step of our algorithm is to report all gapped patterns where both their subpatterns p-matched the text, with a gap of size \( g \) between their occurrences. Hence, for a gap starting at text location \( \ell + 1 \), we consider \( \text{Locc}[\ell], \text{Rocc}[\ell + g + 1] \) and want to report all gapped patterns \( P_i \), where \( \text{prev}(l_{pi}) \) is a suffix of the sequence associated with the state saved at \( \text{Locc}[\ell] \), and \( \text{prev}(r_{pi}) \) is a suffix of the sequence associated with the state saved at \( \text{Rocc}[\ell + g + 1] \), where \( g \in [\alpha_i, \beta_i] \). In the following subsections, we suggest two algorithms for results calculation, the first follows the range query described in [21], enabling solving the pDMOG problem for dictionaries containing variable length gaps while the second follows the second solution of [9], enabling result calculation in \( O(1 + \text{occ}) \) time per a text location and a gap length, where \( \text{occ} \) is the number of reported patterns, yet it solves the pDMOG for dictionaries with a single set of gap boundaries.

5.1 Results Calculation by Rectangle Stabbing

Afshani et. al. [1] considered the problem of Rectangular Stabbing, where a set of \( k \) axis-aligned hyper rectangles are preprocessed, then, given a query \( k \) dimensional point, all \( t \) rectangles that contain the query point, can be easily reported. Note, that by the problem definition, a point on he boundary of a rectangle is not assumed to be contained in the rectangle. They proved the following lemma.

**Lemma 7.** ([1]) A set of \( d \) \( k \)-dimensional rectangles (where \( k \geq 2 \) is a constant) can be preprocessed into \( O(d \log^{k-2} d) \) space data structure which can answer any rectangular stabbing query in \( O(\log^{k-1} d + \text{output}) \)

Hon. et. al. [21] created a hyper rectangle region representing every gapped pattern in the dictionary. When given occurrences of some \( l_{pi} \) and \( r_{pj} \), and a certain gap between the occurrences, they perform a rectangular stabbing query, and report all gapped pattern found in the text according to the given subpatterns and the gap between them. In order to have a single query of \( l_{pi} \) and \( r_{pj} \) and still retrieve all gapped patterns included in the query, that is all gapped patterns \( P_f \) where \( l_{pf} \) is a suffix of \( l_{pi} \) and \( r_{pj} \) is a prefix of \( r_{pj} \), that appear with an appropriate gap between them, they numbered the nodes in the suffix trees they built over the Left and Right subpatterns, by their preorder rank. Such a numbering guarantees that a prefix of some \( r_{pi} \) has a smaller number than \( r_{pi} \) itself. In addition, due to the structure of suffix trees, they had that \( r_{pi} \) was in the subtree of all its prefixes. Therefore, by defining a dimension of the rectangle to be the number of a node in the suffix tree and the rightmost node in its subtree, they obtained the sought after reports.

However, for parameterized patterns, the case is more delicate. We need the rectangle related to a state \( \text{p-labeled by } \text{prev}(l_{pf}) \) to be included in the rectangle related to the state \( \text{p-labeled by } \text{prev}(l_{pf}) \), where \( \text{prev}(l_{pf}) \) is a suffix of \( \text{prev}(l_{pi}) \). Yet, the \( \text{prev} \) function does not preserve the suffix relation of the strings it is applied to. Consider \( x, y \) as two subpatterns, where \( x \) is a suffix of \( y \). It is not guaranteed that \( \text{prev}(x) \) is a suffix of \( \text{prev}(y) \), due to the changes of the \( \text{prev} \) function when deleting characters from the beginning of the string. For example consider \( l_{pi} = uuaa \) and its suffix \( uua \), so \( \text{prev}(l_{pi}) = 011a \) yet, \( \text{prev}(uua) = 01a \), which is not a suffix of \( 011a \).

Nevertheless, in the p-AC automaton, we can find the suffix of a p-subpattern by its \( \text{pfail} \) link, as it points to a prefix of a p-subpattern that is a suffix of the
p-subpattern p-labeling the current state. Therefore, we construct for LpAC the trie Lpfail and for RpAC the trie Rpfail respectively, where the nodes of the trie are the states of the p-automaton, the root of the trie corresponds to the start state of the automaton and the children of a node x are all the states having a pfail link to x in the p-automaton. Obviously, the construction of these tries is done in linear time in the size of the p-automata. Numbering the nodes of Lpfail, Rpfail by their preorder rank, yields the possibility to use the rectangular stabbing procedure efficiently for parameterized gapped dictionaries.

In the preprocess, we number each state x of LpAC according to its preorder number in Lpfail and denote it by lnump(x). Similarly rnum(y) is the preorder number of state y of RpAC in the trie Rpfail. We name an LpAC state, p-labeled by prev(lp1) by lstateip1, and the RpAC state p-labeled by prev(rp1) is named rstateip1. Then, for every gapped pattern 

\[ P \] 

region \( R \) in \( D \) we construct a hyper rectangular region \( R \) in 3D where \( R_i = [\text{lnump}(lstate_{ip1}) - 1, \text{lnump}(x) + 1] \times [\text{rnum}(rstate_{ip1}) - 1, \text{rnum}(y) + 1] \times [\alpha_i - 1, \beta_i + 1] \) where \( x \) is the rightmost leaf node in the subtree of lstate_{ip1} in Lpfail, \( y \) is the rightmost leaf node in the subtree of rstate_{ip1} in Rpfail and \( \alpha_i, \beta_i \) are the gap boundaries of \( P_i \).

**Lemma 8.** Given the filled Locc, Rocc arrays, performing a Rectangular Stabbing query of point \( (\text{lnump}(\text{Locc}[\ell]), \text{rnum}(\text{Rocc}[\ell + g + 1]), g) \) for \( \alpha_{\min} \leq g \leq \beta_{\max} \) where \( \alpha_{\min} = \min_{1 \leq i \leq d}\{\alpha_i\} \), \( \beta_{\max} = \max_{1 \leq i \leq d}\{\beta_i\} \), yields all gapped patterns \( P \), p-matching text \( T \), such that the occurrence of \( \text{prev}(lp_i) \) ends at \( \text{prev}(T[\ell]) \) and there is a beginning of an occurrence of \( \text{prev}(rp_i) \) after a gap of \( g \) characters.

Such a query requires \( O(\log^2 d + \text{occ}) \) time and space of \( O(d \log d) \), where \( d \) is the number of gapped patterns and \( \text{occ} \) is the number of patterns reported as output.

**Proof.** Given the query point \( (\text{lnump}(\text{Locc}[\ell]), \text{rnum}(\text{Rocc}[\ell + g + 1]), g) \), according to [1] all \( R_i = [a, a'] \times [b, b'] \times [c, c'] \) are retrieved, where \( a < \text{lnump}(\text{Locc}[\ell]) < a' \), \( b < \text{rnum}(\text{Rocc}[\ell + g + 1]) < b' \) and \( c < g < c' \) holds. Suppose some \( \text{prev}(lp_i) \) was located ending at location \( \ell \) and \( \text{prev}(rp_i) \) was located ending at location \( \ell + g + 1 \) in \( T^R \), thus the query point is \( (\text{lnump}(lstate_{ip1}), \text{rnum}(rstate_{ip1}), g) \). Obviously \( \text{lnump}(lstate_{ip1}) - 1 < \text{lnump}(lstate_{ip1}) < \text{lnump}(x) + 1 \) and \( \text{rnum}(rstate_{ip1}) - 1 < \text{rnum}(rstate_{ip1}) < \text{rnum}(y) + 1 \), when \( x, y \) are the rightmost leaves in the subtrees of lstate_{ip1}, rstate_{ip1} in Lpfail, Rpfail respectively, due to the preorder numbering. Hence, \( R_i \) is stabbed and \( P_i \) is reported if the gap length \( g \) between the subpatterns, is in accordance with boundaries \( \alpha_i \) and \( \beta_i \).

Another possible case is that \( \text{Locc}[\ell] = f \), \( \text{Rocc}[\ell + g + 1] = h \) and \( \text{prev}(lp_i) \) is a suffix of the p-label of state \( f \) and \( \text{prev}(rp_i) \) is a suffix of the p-label of state \( h \). Thus \( P_i \) needs to be reported in case the gap fits. Since \( \text{prev}(lp_i) \) is a suffix of the p-label of state \( f \), it follows that the state p-labeled by \( \text{prev}(lp_i) \) is an ancestor of state \( f \) in the Lpfail trie, thus \( \text{lnump}(lstate_{ip1}) < \text{lnump}(f) \) due to the preorder numbering. Moreover, as \( f \) is included in the subtree rooted by lstate_{ip1}, we have that \( \text{lnump}(f) < \text{lnump}(\text{the rightmost leaf in the subtree rooted by lstate}_{ip1}) + 1 \). Similarly we have that \( \text{rnum}(rstate_{ip1}) < \text{rnum}(h) \) and \( \text{rnum}(h) < \text{rnum}(\text{the rightmost leaf in the subtree rooted by rstate}_{ip1}) + 1 \). It follows that the hyper rectangle \( R_i \) is stabbed by the query point, if the gap of length \( g \) between the located subpattern is in accordance with boundaries \( \alpha_i \) and \( \beta_i \), thus \( P_i \) is reported.

The time and space complexity of a query follow Lemma [7] considering the case of \( d \) hyper rectangles in 3D, constructed in the preprocess. □
We perform such rectangular stabbing queries, for every text location \(1 \leq \ell \leq n\) and for every possible gap size, \(\alpha_{\text{min}} \leq g \leq \beta_{\text{max}}\) where \(\alpha_{\text{min}} = \min_{1 \leq i \leq d}\{\alpha_i\}\), \(\beta_{\text{max}} = \max_{1 \leq i \leq d}\{\beta_i\}\).

Lemma \(\Box\) and Lemma \(\Box\) yields Theorem \(\Box\).

**Theorem 9.** The pDMOG problem for dictionary \(D\) with variable length gaps and text query \(T\), can be solved in \(O(|D| \log |D| + n)\) space, and with a query time of \(O(n(\beta_{\text{max}} - \alpha_{\text{min}}) \log^2 d + \text{occ})\), where \(n\) is the size of \(T\), \(d\) is the number of gapped patterns in the dictionary and \(\text{occ}\) is the number of reported patterns.

### 5.2 Results Calculation by Look-up Table

In case all gapped patterns share their gap boundaries and a query time is crucial, we suggest solving the intersection between the appearances of p-subpatterns using a lookup table named \(\text{out}\), though it implies an increase in preprocessing time.

For an efficient filling of the lookup table, the subpatterns numbering has to satisfy the rule that the longer a subpattern, the higher its numbering, that is, \(\lnum(lstate_{lp_\ell}) > \lnum(lstate_{lp_{\ell + 1}})\), (where \(lstate_{lp_\ell}\) is the state p-labeled by \(\text{prev}(\ell lp_\ell)\) in \(LpAC\)) iff \(|lp_\ell| \geq |lp_{\ell + 1}|\). Similarly, \(\lnum(rstate_{rp_\ell}) > \lnum(rstate_{rp_{\ell + 1}})\), (where \(rstate_{rp_\ell}\) is the state p-labeled by \(\text{prev}(\ell rp_\ell)\) in \(RpAC\)), iff \(|rp_\ell| \geq |rp_{\ell + 1}|\). The numbering system from the previous subsection can be used as well as a simple BFS traversal over \(LpAC/RpAC\).

The look-up table consists of accepting states, yet, \(\text{Locc}[\ell]\) and \(\text{Rocc}[\ell + g + 1]\) can include any state in each of the p-automata, thus for each state \(x\) in each of the p-automata, that is not an accepting state, we save \(\text{accept}(x)\) that is the accepting state with the longest p-label that is a suffix of the p-label of \(x\). The \(\text{accept}(x)\) are calculated, by a BFS traversal over the automaton. When reaching state \(x\) that is not an accepting state, we consider its pfail link, where \(\text{pfail}(x)\) points to the longest p-labeled state that its p-label is a suffix of the p-label of \(x\). In case \(\text{pfail}(x)\) is an accepting state, then \(\text{accept}(x) = \text{pfail}(x)\), otherwise \(\text{accept}(x) = \text{accept}(\text{pfail}(x))\).

For every accepting state \(lstate_{lp_\ell} \in LpAC\) we save a link \(\text{psuf}(lp_\ell)\) that leads to the \(\lnum\) of an accepting state p-labeled by the longest \(lp_k\) such that \(\text{prev}(lp_k)\) is a real suffix of \(\text{prev}(lp_\ell)\), if it exists. We define \(\text{psuf}(x) = \lnum(\text{pfail}(x))\) if \(\text{pfail}(x)\) is an accepting state and \(\text{psuf}(x) = \lnum(\text{accept}(\text{pfail}(x)))\) otherwise. Similarly, for every accepting state \(rstate_{rp_\ell} \in RpAC\), we save a link \(\text{psuf}(rp^R_\ell)\) that leads to the \(\lnum\) of an accepting state p-labeled by the longest \(rp_k\) such that \(\text{prev}(rp^R_k)\) is a real suffix of \(\text{prev}(rp^R_\ell)\), if it exists. This link is similarly calculated in the \(Rpfail\) trie.

The \(\text{out}\) table is of size \(d_{\text{left}} \times d_{\text{right}}\). Entry \(\text{out}[f, h]\) refers to the set of all indices of gapped patterns that are reported when \(\text{prev}(lp_\ell)\) is the longest subpattern that appears at the suffix of \(\text{prev}(T[1 \ldots \ell])\) and \(\lnum(lstate_{lp_\ell}) = f\), and when \(\text{prev}(rp^R_\ell)\) is the longest subpattern that appears at the suffix of \(\text{prev}(T[n \ldots \ell + g + 1])\), and \(\lnum(lstate_{lp_\ell}) = h\) and \(\alpha \leq g \leq \beta\). The \(\text{out}\) table is recursively filled in increasing order of indices, where filling \(\text{out}[f, h]\) entry implies filling four fields:

1. Index field, \(\text{out}[f, h].\text{index} = i\) iff \(i = j\). (Note that at most one index can be saved at \(\text{out}[f, h].\text{index}\) as two patterns are bound to differ by at least one subpattern, having a single set of gap boundaries.)
2. up link, where \(\text{out}[f, h].\text{up} = [f', h]\) iff \(\text{prev}(lp_k)\) is the longest suffix of \(\text{prev}(lp_\ell)\) where \(f' = \lnum(lstate_{lp_\ell})\) and \(k = j\).
3. *left* link, where \( \text{out}[f, h].\text{left} = [f, h'] \) iff \( \text{prev}(\text{rp}_k^R) \) is the longest suffix of \( \text{prev}(\text{rp}_k^R) \) where \( h' = \text{rnum}(\text{rstate}_{\text{rp}_k}) \) and \( k = i \).

4. *back* link, where \( \text{out}[f, h].\text{back} = [\text{psuf}^*(f), \text{psuf}^*(h)] \), where \( \text{psuf}^*(f) \) is the longest real suffix of \( \text{prev}(\text{lp}_i) \) and \( \text{psuf}^*(h) \) is the longest real suffix of \( \text{prev}(\text{rp}_k^R) \) such either \( \text{psuf}^*(f) \) or \( \text{psuf}^*(h) \) form a gapped pattern with the suffix of the other. (\( \text{psuf}^*(f) \) can be obtained by recursively applying the \( \text{psuf} \) links.)

The lookup table is filled by the following formal recursive rule.

**The Recursive Rule**

\[
\text{out}[f, h].\text{up} = \begin{cases} 
[\text{psuf}(f), h] & \text{if } \text{out}[\text{psuf}(f), h].\text{index} \neq \text{null} \\
\text{out}[\text{psuf}(f), h].\text{up} & \text{otherwise}
\end{cases}
\]

\[
\text{out}[f, h].\text{left} = \begin{cases} 
[f, \text{psuf}(h)] & \text{if } \text{out}[f, \text{psuf}(h)].\text{index} \neq \text{null} \\
\text{out}[f, \text{psuf}(h)].\text{left} & \text{otherwise}
\end{cases}
\]

\[
\text{out}[f, h].\text{back} = \begin{cases} 
\text{if } \text{out}[\text{psuf}(f), \text{psuf}(h)].\text{index} \neq \text{null} \\
\text{out}[\text{psuf}(f), \text{psuf}(h)].\text{back} & \text{otherwise}
\end{cases}
\]

\[
\text{if } \text{out}[\text{psuf}(f), \text{psuf}(h)].\text{index} \neq \text{null} \\
\text{or } \text{out}[\text{psuf}(f), \text{psuf}(h)].\text{up} \neq \text{null} \\
\text{or } \text{out}[\text{psuf}(f), \text{psuf}(h)].\text{left} \neq \text{null}
\]

Considering \( \text{Locc}[\ell] = f, \text{Rocc}[\ell + g + 1] = h \), the results calculation is performed by consulting entry \( \text{out}[f, h] \). We report \( \text{out}[f, h].\text{index} \) if it exists, yet in order to report all relevant patterns, that their \( p \)-subpatterns are numbered by \( f \) or by \( h \) or that they are suffixes of the \( p \)-label of the states numbered by \( f, h \), we follow the links saved at \( \text{out}[f, h] \), as detailed in the procedure:

**ResultsQuery( \( f, h \) )**

1. If \( \text{out}[f, h].\text{index} \neq \text{null} \), report \( \text{out}[f, h].\text{index} \).
2. If \( \text{out}[f, h].\text{back} \neq \text{null} \)
   - **ResultsQuery( \( f', h' \) ) for \( [f', h'] = \text{out}[f, h].\text{back} \).
3. Let \( f' \leftarrow f, h' \leftarrow h \).
4. While \( (\text{out}[f', h'].\text{up} \neq \text{null}) \).
   - (a) Let \( [f', h] = \text{out}[f', h].\text{up} \).
   - (b) Report \( \text{out}[f', h].\text{index} \).
5. While \( (\text{out}[f, h'].\text{left} \neq \text{null}) \).
   - (a) Let \( [f, h'] = \text{out}[f, h'].\text{left} \).
   - (b) Report \( \text{out}[f, h'].\text{index} \).

**Lemma 10.** The procedure ResultsQuery, given the gapped dictionary \( D \), \( \text{Locc}[\ell] = f, \text{Rocc}[\ell + g + 1] = h \) and the \( \text{psuf} \) function, reports all dictionary patterns appearing with gap of size \( g \) starting at \( T[\ell + 1] \).

**Proof.** Due to the construction of the p-AC automata, we have that state numbered by \( f \) represents \( \text{prev}(\text{lp}_i) \) and all its suffixes and the state numbered \( h \) represents a certain \( \text{prev}(\text{rp}_k^R) \) and all its prefixes. According to the AC algorithm the subpatterns represented by these states are of maximal length [2].

In order to report all required patterns, entry \( \text{out}[f, h] \) for \( 1 \leq f \leq d_{\text{left}}, 1 \leq h \leq d_{\text{right}} \), has to contain links to all entries containing indices of patterns whose \( \text{left} \) subpattern is represented by the state numbered \( f \) and its \( \text{right} \) subpattern is represented by the state numbered \( h \). There are 4 possible cases:
These observations, can be easily proved by induction. □
Lemma 11. The construction of the out table requires \( O(|D| \log |D| + d_{left} \times d_{right}) \) time and \( O(d_{left} \times d_{right}) \) space. Performing a query on the out table regarding \( p \)-subpatterns appearing adjacently to a gap starting at \( T[\ell + 1] \), requires \( O(1 + \text{occ}) \) time, where \( \text{occ} \) is the number of patterns reported.

Proof. The preprocess requires numbering the accepting states of both \( p \)-AC automata and computing the accept and psuf links, all can be done by performing a BFS traversal over \( p \)-automata and the \( Lpfail, Rpfail \) tries, in linear time in the size of the \( p \)-automata and tries, \( O(|D| \log |D|) \). Filling \( dout[f, h] \) entries with index \( i \) when \( lnum(lstate_{lp_i}) = f \) and \( rnum(rstate_{rp_i}) = h \), can be done in \( O(d) \) time. Filling each of the entries of the table \( dout[f, h] \) can be performed in \( O(1) \) by the recursive rule.

The table query procedure is based on following links and reporting indices found. Every step of following an \( up \) or \( left \) link implies that the linked entry contains a pattern index, needs to be reported. The \( back \) link either directs us to an entry including a pattern index, needs to be reported or it directs us to an entry containing an \( up \) or \( left \) links. Hence, by following at most two links we encounter an index needs to be reported. Consequently, the time of following links is attributed to the size of the output.

The lookup table has \( d_{left} \times d_{right} \) entries, each consists of 4 fields, yielding \( O(d_{left} \times d_{right}) \) space requirement.

\[\square\]

For every text location \( 1 \leq \ell \leq n \) and for every possible gap size, \( \alpha \leq g \leq \beta \) we perform a look up table query \( out[lnum(\text{Locc}[\ell]), rnum(\text{Rocc}[\ell + g + 1])] \).

Lemma 6 and Lemma 11 yields Theorem 12.

Theorem 12. The \( pDMOG \) problem for dictionary \( D \) with a single set of gap boundaries and text query \( T \), can be solved in \( O(|D| \log |D| + d^2) \) space and with query time \( O(n(\beta - \alpha) + \text{occ}) \), where \( n \) is the size of the text and \( \text{occ} \) is the number of reported gapped patterns.

6 Conclusions and Open Problems

This paper suggests the problem of dictionary matching with one gap where the matching technique is parameterized, a problem with tight relation to cyber security. The paper presents efficient and simple to program algorithms.

There are several interesting open problems related to the \( pDMOG \) problem, such as solving the \( DMOG \) problem for other methods of encrypted gapped patterns and solving the \( pDMG \) for patterns containing multiple gaps. Since the \( DMOG \) problem is a crucial bottleneck procedure in network intrusion detection system applications, these open problems should be addressed in the future.
References


