Paralelismo em Buscas Criptografadas Verificáveis*

Luis A. B. Pacheco[1], Eduardo Alchieri[1] e Priscila Solis Barreto[1]

[1]Departamento de Ciência da Computação
Universidade de Brasília (UnB)
Brasília – DF – Brasil

{luis.pacheco,alchieri,pris}@unb.br

Abstract. *Although the cloud computing model is being widely used, one of the factors limiting its adoption for many companies and institutions is related to ensuring the security and privacy of information stored in the cloud. In this context, searchable encryption allows the execution of searches on encrypted data without reveal sensitive data to the servers. Searchable encryption algorithms are generally designed to execute sequential searches by a single core, but current servers provide multiple cores to applications. In this work we propose two algorithms for searchable encryption, aiming to allow both execution parallelism and reuse of computations among searches. An experimental analysis shows the advantages of the proposed solutions.*

Resumo. *Apesar do modelo de computação em nuvem estar sendo amplamente utilizado, um dos fatores que limita sua adoção para muitas empresas e instituições está relacionado com a garantia de segurança e privacidade das informações armazenadas na nuvem. Neste contexto, criptografia buscável possibilita a realização de buscas sobre dados criptografados sem que os servidores acessem estes dados sensíveis. Algoritmos para criptografia buscável geralmente são projetados para executar buscas de forma sequencial por um único núcleo de processamento, enquanto que os servidores atuais disponibilizam vários núcleos para as aplicações. Neste trabalho propomos dois algoritmos para buscas criptografadas, com o objetivo de permitir tanto o paralelismo na execução quanto o reaproveitamento de computações entre as buscas. Os resultados experimentais mostram que os dois algoritmos propostos aumentam o desempenho das buscas em mais de 95% em relação ao algoritmo sequencial.*

1. Introdução

A utilização de plataformas de Computação em Nuvem está disseminada na conjuntura atual, uma vez que usuários domésticos, organizações privadas e públicas utilizam a computação em nuvem em cenários diferentes. No entanto, ainda existem muitos desafios nesta área, como por exemplo assegurar os requisitos de privacidade de usuários e entidades que terceirizam seus recursos computacionais a a outras organizações. Neste âmbito, é de grande importância que as plataformas de nuvem ofereçam meios de armazenar dados sensíveis, os quais podem ser acessados apenas pelas partes autorizadas, de tal forma que a própria plataforma tenha acesso limitado. Exemplos deste tipo de cenário incluem dados médicos e governamentais.

*Trabalho parcialmente financiado pelo MCTIC/RNP/CTIC através dos projetos Atmosphere e P4Sec.

Em cenários onde nem mesmo a plataforma de nuvem deve ter acesso aos dados do usuário, a solução adotada atualmente é o armazenamento de dados criptografados. No entanto, este recurso dificulta o acesso a tais dados, pois os mecanismos de busca não possuem acesso ao conteúdo, e portanto não podem ser utilizados. Criptografia Buscável (CB) (*Searchable Encryption* – SE) possibilita a busca em dados criptografados, aumentando a viabilidade da utilização de Computação em Nuvem para dados sensíveis [Wang et al. 2016]. De uma forma geral, os documentos são armazenados nos servidores de forma criptografada e um índice criptografado é criado pelo usuário para relacionar palavras-chave aos documentos. Este índice é usado nos servidores para permitir que buscas, também criptografadas pelos usuários, sejam associadas aos respectivos documentos buscados. Neste esquema os servidores não conseguem acessar nenhuma informação, i.e., documentos, índice ou dados das buscas.

Um mecanismo de CB permite que um servidor pesquise em dados criptografados revelando a menor quantidade de informações possíveis. Em geral, algoritmos de CB ponderam entre três fatores: *eficiência* – relacionada ao custo computacional necessário para que a busca seja realizada; *expressividade* da consulta – referente a funcionalidades de busca; e *segurança* – define as informações vazadas para a plataforma de nuvem para permitir a execução das buscas. Em geral, uma maior expressividade da consulta resulta em mais informações vazadas para a plataforma, ao mesmo tempo que primitivas de segurança mais robustas impõe um maior custo computacional [Poh et al. 2017].

Em relação à eficiência, muitos mecanismos de CB alcançam tempos sublineares para execução das buscas, como $O(log\ w)$, e até mesmo ótimos ($O(|D(w)|)$), em que w é a quantidade de palavras-chave e $|D(w)|$ é a quantidade de documentos contendo a palavra-chave w. Em geral, estes tempos são alcançados pelo uso de índices invertidos e árvores de *hash*, que mapeiam as palavras-chave aos documentos que contém tais palavras. Estes algoritmos geralmente são projetados para executarem buscas de forma sequencial, utilizando apenas um núcleo de processamento. No entanto, as plataformas atuais de computação em nuvem permitem a utilização de recursos que contém vários núcleos de processamento. A quantidade de núcleos disponíveis inclusive pode ser alterada durante a execução devido à propriedade de elasticidade destas plataformas. Neste trabalho, o foco é aumentar o desempenho destas soluções através de algoritmos que possibilitam a execução paralela de partes das buscas nos múltiplos núcleos disponíveis.

Neste âmbito, o algoritmo *Verifiable Multi-Keyword Ranked Search* (VMRS) [Jiang et al. 2017] propõe um mecanismo de criptografia buscável que suporta múltiplas palavras, verificação dos resultados pelo usuário e classificação dos resultados, de acordo com a relevância relacionada com as palavras buscadas. Em especial, o VMRS utiliza uma estrutura baseada em índice invertido chamada QSet, que utiliza criptografia baseada em emparelhamento bilinear [Boneh and Franklin 2001], a qual aumenta consideravelmente as propriedades de segurança, no entanto com um grande custo computacional.

Este trabalho tem como objetivo aumentar o desempenho das buscas realizadas sobre dados criptografados através da execução paralela de partes da busca. As soluções propostas foram construídas baseando-se no VMRS, por este ser um algoritmo que fornece funcionalidades interessantes para o usuário como as anteriormente descritas. Resumidamente, este trabalho apresenta as seguintes contribuições:

- Proposta do pVMRS (*Parallel* VMRS), que introduz paralelismo de forma isolada

em uma única busca. Devido ao grande custo computacional da criptografia de emparelhamento, estas operações são paralelizadas nos núcleos disponíveis.
- Proposta do bVMRS (*Batched* VMRS), que além de paralelizar as operações de emparelhamento, agrega buscas em lotes e utiliza memorização para reduzir a quantidade de operações realizadas. A ideia geral deste algoritmo é reaproveitar computações já realizadas entre as buscas contidas em um lote.
- Análise experimental que mostra as vantagens das soluções propostas em diferentes cenários de buscas.

O restante deste trabalho é organizado como segue. A Seção 2 discute os trabalhos relacionados. A Seção 3 detalha os algoritmos propostos. A Seção 4 apresenta experimentos, mostrando características do pVMRS e bVMRS, e a Seção 5 conclui o trabalho.

2. Trabalhos Relacionados

Soluções de criptografia buscável são baseadas em criptografia simétrica ou assimétrica. Algoritmos baseados em criptografia simétrica permitem que usuários em posse da chave secreta possam gerenciar os arquivos pertencentes ao índice e realizar buscas, enquanto que algoritmos de criptografia assimétrica permitem que usuários em posse da chave pública adicionem arquivos ao índice, mas apenas o usuário em posse da chave privada realize a busca no índice [Wang et al. 2016]. Nos algoritmos de criptografia simétrica é necessário um elemento responsável por gerenciar a distribuição das chaves secretas.

Este trabalho aborda a paralelização em algoritmos baseados em criptografia simétrica. Em geral, um índice mascarado é construído pelo usuário e enviado ao servidor de nuvem. Este índice permite que palavras-chave mascaradas (*trapdoor*) sejam associadas aos arquivos que as possuem. Trabalhos que propõem mecanismos de criptografia buscável implementam, de forma geral, as funções *CriarÍndide*, *Criptografar* e *Buscar* [Poh et al. 2017]. A função *CriarÍndide* cria um índice criptografado buscável, este índice pode ligar documentos a palavras-chave ou o contrário, neste último caso é chamado de índice invertido. A função *Criptografar* criptografa os dados a serem armazenados na nuvem. A função *Buscar*, executada pela plataforma de nuvem, retorna os dados criptografados correspondentes a uma ou mais palavras-chave mascaradas. Apesar de se desejar que todas estas funções tenham bom desempenho, note que a busca é a mais importante uma vez que é executada diversas vezes nos servidores.

Algoritmos de criptografia buscável implementam diversas funcionalidades de acordo com sua finalidade, como [Poh et al. 2017]: busca por palavra única – a consulta permite que apenas uma palavra-chave seja buscada, o servidor retorna os documentos que possuem tal palavra; busca difusa – o mecanismo é capaz de determinar a proximidade entre as palavras-chave, podendo retornar não apenas documentos que possuem a palavra, mas também documentos que possuem palavras parecidas; busca conjuntiva – permite a consulta por mais de uma palavra-chave por vez; busca classificada – retorna uma quantidade de documentos classificada, indicando qual documento possui maior relevância à consulta realizada; e busca verificável – possibilita que o usuário inspecione que o servidor está retornando os documentos corretos. É importante notar que certas funcionalidades podem vazar informações indesejadas ao servidor. Por exemplo, na busca difusa o servidor é capaz de determinar a proximidade entre os palavras, mesmo elas estando mascaradas por algum algoritmo de *hash*. Portanto, existe um custo benefício claro entre funcionalidades e o nível de privacidade alcançada.

Com relação à segurança, diversas primitivas são utilizadas para assegurar a privacidade dos dados do usuário. Nos algoritmos baseados em criptografia simétrica, as principais primitivas são as funções pseudo-aleatórias (*pseudo-random functions* – PRF) [Stallings et al. 2012], permutações pseudo-aleatórias (*pseudo-random permutations* – PRP) [Stallings et al. 2012] e esquemas de criptografia simétrica [Poh et al. 2017]. Além destas primitivas bastante difundidas, mecanismos de áreas relacionadas são empregados com frequência, alguns exemplos são: criptografia homomórfica [van Dijk et al. 2010], criptografia com preservação de ordem [Boldyreva et al. 2009], criptografia baseada em emparelhamento bilinear [Boneh and Franklin 2001], e *Oblivious RAM* (ORAM) [Pinkas and Reinman 2010].

Cash et al [Cash et al. 2013] propuseram um mecanismo de busca baseado em criptografia simétrica que suporta busca conjuntiva escalável para conjuntos de dados grandes. Este foi o primeiro trabalho a suportar buscas conjuntivas em tempo sublinear. O algoritmo se baseia em um índice invertido, em que é possível determinar os dados que possuem a primeira palavra da busca, e então verificar se estes dados também possuem as palavras restantes. O algoritmo é construído de modo que apenas o resultado da busca é revelado ao servidor, ou seja, apenas dados presentes que contêm todas as palavras e não os dados que contêm algumas das palavra.

Chen et al [Chen et al. 2016] implementam um índice hierárquico, o qual agrupa os documentos de acordo com sua relevância. O mecanismo possui busca conjuntiva e classificada. O algoritmo atinge eficiência ao limitar a busca apenas a grupos de documentos que possuem alta similaridade com a consulta realizada. O algoritmo é dinâmico, ou seja, permite a edição do índice sem precisar recriá-lo. O tempo de busca aumenta linearmente enquanto que o tamanho do conjunto de dados cresce exponencialmente.

Em [Jiang et al. 2017] é proposto um algoritmo de criptografia buscável para múltiplas palavras, que é verificável e classificável. O algoritmo utiliza uma estrutura de índice invertido criptografado (abordagem bastante comum em trabalhos desta área), desta forma a busca é realizada em $O(log(n))$, onde n é a quantidade de documentos que possuem a intersecção das palavras buscadas. A classificação dos documentos encontrados é realizada pela relevância do documento em relação às palavras buscadas, o valor é calculado tendo em consideração que palavras contidas em menos documentos são mais importantes.

O índice invertido criptografado indexa palavras-chave a uma lista de identificadores dos documentos que contem a palavra. A computação da intersecção de múltiplas-palavras em um documento é realizada por uma estrutura chamada QSet, que indexa a tupla $(palavra, documento)$ ao valor de relevância da palavra no documento. Desta forma, o algoritmo primeiramente encontra os documentos que possuem a primeira palavra no índice invertido, e então verifica, através do QSet, se estes documentos possuem as outras palavras buscadas. Ambas as estruturas são criptografadas por algoritmos PRF e MAC para que a plataforma de nuvem não tenha conhecimento das palavras buscadas. O resultado da busca inclui um código MAC criptografado para cada par de palavra buscada e documento encontrado, que apenas o usuário pode descriptografar. Dessa forma, é possível verificar se todos os documentos retornados pelo provedor de nuvem de fato incluem as palavras buscadas. Os autores provam que o algoritmo proposto é IND-CKA2 (*adaptive indistinguishability under chosen keyword attacks*), ou seja, a plataforma de

nuvem tem acesso apenas ao padrão de acesso e ao padrão de busca.

3. Paralelismo em Buscas Criptografadas Verificáveis

Esta seção descreve os algoritmos propostos para buscas criptografadas com execuções paralelas e em lotes, além das técnicas de implementação utilizadas para alcançar os objetivos deste estudo. Os algoritmos desenvolvidos foram projetados para plataformas heterogêneas, como CPUs de múltiplos núcleos. Em geral, os algoritmos realizam um pré-processamento, coletam as informações e criam as estruturas de dados necessárias para então realizar a execução das operações mais custosas em paralelo.

Antes de detalhar os algoritmos propostos, a seção seguinte apresenta as ideias principais relacionadas com o algoritmo VMRS sequencial (tradicional) para então apresentar as soluções com execuções paralelas de partes da busca (pVRMS) e com reaproveitamento de computações entre buscas agrupadas em lotes (bVRMS).

3.1. VMRS Sequencial

O VMRS sequencial [Jiang et al. 2017] é modelado através de 3 entidades: o proprietário dos dados, o usuário dos dados e a plataforma de nuvem. O proprietário criptografa e envia seus documentos a plataforma de nuvem, junto com um índice também criptografado. A plataforma armazena os documentos e o índice, e recebe consultas criptografadas dos usuários para realizar buscas no índice. A classificação dos documentos é realizada através de uma pontuação de relevância chamada $TF\ x\ IDF$, utilizada amplamente [Cao et al. 2014]. Este cálculo estatístico considera a quantidade de aparições das palavras em cada documento do conjunto de dados (TF) pela quantidade de documentos que possuem tal palavra (IDF).

Considerando uma busca $B = \{w_1, w_2, w_3, ..., w_n\}$. Para fornecer privacidade, é proposto um índice com 3 estruturas distintas, apresentadas na Figura 1. A partir de um dicionário D de palavras e um conjunto de identificadores dos documentos, $F = \{f_{id1}, f_{id2}, f_{id3}, .., f_{idn}\}$, é criado um vetor para cada palavra em D que contém todos os identificadores de F que possuem tal palavra. A matriz representada por todos os vetores é chamada de *Array List* (AL). O *QSet* é um mapa de chave /valor em que a chave é a concatenação de cada elemento de F e palavra w contida neste documento. As palavras são indexadas em outra estrutura de chave /valor, chamada *Lookup Table* (LT).

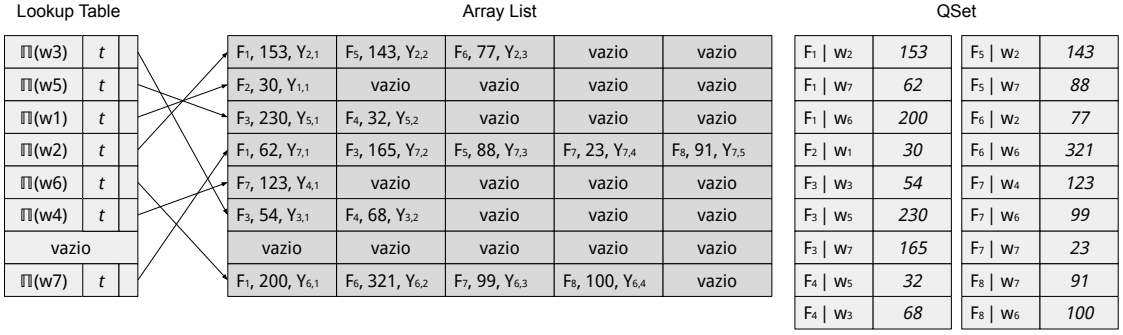


Figura 1. Estruturas do VMRS sequencial.

A chave da LT é dada por $\pi(w)$, em que π é uma PRF e w é uma palavra de D. A LT também armazena um vetor autenticado t, contendo todos os documentos que possuem

determinada palavra, utilizado na verificação dos resultados pelo usuário. As chaves do *QSet* são protegidas por uma técnica chamada *blinded exponentiation* em um grupo G de ordem de primos p. Para cada par f_{id}/w a chave é calculada por $g^{\psi_{k3}(w).\psi_{k1}(f_{id})}$, em que ψ é uma PRF em Z_p^*. O valor da entrada é a relevância de w em relação à f_{id} criptografada. Na entrada do *Array List* é armazenado $y_{i,j} = \psi_{k1}(f_j).z_c^{-1}$, em que z_c corresponde a $\psi(w_i||c)$, e c é um contador relativo a cada entrada no AL. Esta abordagem protege o índice contra ataques de busca cruzada, em que o servidor pode utilizar elementos de consultas distintas para descobrir novas informações.

Os elementos do *Array List* possuem o f_{id}, a pontuação de relevância e $y_{i,j}$. Estas informações são agrupadas e também criptografadas. Tanto a LT quanto o AL são preenchidos com entradas falsas, evitando que o servidor identifique a quantidade de palavras no dicionário e a quantidade de documentos que possuem determinada palavra.

De forma resumida, para realizar uma busca o usuário deve enviar $\pi(w_1)$ e k, onde k é a quantidade de documentos a serem retornados, além de outros elementos necessários para que o servidor encontre os resultados. Ao receber a busca, o servidor acessa os elementos apontados por $\pi(w_1)$ em LT. Para cada elemento (que representa um documento) é verificado se $w_2, ..., w_n$ está nesse documento, para tanto é calculada a chave do *QSet*. Para gerar a chave do *QSet* para cada f_j de w_i o servidor deve solicitar um $token$ ao usuário, em que $token[c, i] = g^{\psi_k 3(w_i).z_c}$ e então calcular $token[c, i]^{y_{1,c}}$. Esta operação de criptografia baseada em emparelhamento bilinear é a mais custosa do algoritmo. O servidor então retorna os k documentos que possuem entradas em *QSet* para todas as palavras da busca. O usuário verifica se os resultados são válidos caso todos os documentos retornados estejam em t.

3.2. pVMRS: *Parallel* VMRS

A partir de experimentos iniciais realizados com o algoritmo VMRS sequencial, foi verificado que a operação de maior custo no algoritmo é o emparelhamento bilinear realizado para produzir a chave do QSet, necessária para verificar a existência das palavras da busca nos arquivos da primeira palavra. Felizmente esta operação, que possui apenas dois parâmetros de entrada e produz a chave do QSet como saída, pode ser convenientemente paralelizada. Os dois parâmetros de entrada são (i) o *token* relativo a posição do arquivo no *array* da primeira palavra e à palavra a ser buscada, fornecido pelo usuário ao realizar a busca, e (ii) o identificador (id) do arquivo a ser buscado, criptografado por uma PRF.

O algoritmo pVRMS proposto permeia apenas a busca a ser realizada no índice gerado pelo usuário, o qual foi modificado para otimizar a busca. Ao invés de utilizar um bit para identificar o último arquivo na lista de arquivos de uma palavra, a quantidade de arquivos é armazenada na *Lookup Table*. Esta modificação facilita a paralelização, pois desta forma não é necessário percorrer a lista de forma sequencial para identificar seu tamanho. O índice com esta modificação é utilizado também no algoritmo bVRMS apresentado na seção seguinte.

O pVRMS proposto visa diminuir o tempo necessário para realizar uma única busca, i.e., a paralelização ocorre dentro da execução da busca. Conforme comentado, identificou-se que as operações de emparelhamento bilinear são as mais custosas e podem ser realizadas em paralelo, mas para tanto algum pre-processamento deve ser realizado para preparar as informações e estruturas necessárias. De forma geral o algoritmo é

executado em quatro etapas (Algoritmo 1):

Algorithm 1: Algoritmo pVMRS

entrada: Lookup Table lt, ArrayList al, Keyset ks, busca b, QSet qs e quantidade de arquivos buscados k
saída : Lista de arquivos encontrados

Etapa 1

```
// Inicialização e busca da primeira palavra na lt
w1 ⟵ b[0]
qtd_outras_palavras ⟵ (quantidade de palavras em b) − 1
qtd_arquivos ⟵ lt[w1].tamanho
indice_w1 ⟵ lt[w1].end
arqs_w1 ⟵ al[indice_w1]
Y_{i,c} ⟵ ∅
Arqs ⟵ ∅
Pontos ⟵ ∅
Resultado ⟵ ∅
```

Etapa 2

```
// Decodifica cada entrada do array list para w1
for i ← 1 to |arqs_w1| do
    Y_{i,c}[i] ← decodifica de arqs_w1[i]
    Arqs[i] ← decodifica de arqs_w1[i]
    Pontos[i] ← decodifica de arqs_w1[i]
end
```

Etapa 3

```
// Recupera a pontuação no QSet para cada outra palavra e cada
   arquivo
for i ← 1 to qtd_arquivos do
    for j ← 1 to qtd_outras_palavras do
        token ← SolicitaToken(Arqs[i], b[j])
        indice_qset ← token^{Y_{i,c}[i]}
        if qs[indice_qset] existe then
            Inicio SecaoCritica
                Pontos[i] ← Pontos[i] + qs[indice_qset]
            Fim SecaoCritica
        end
    end
end
```

Etapa 4

```
// Ordena e filtra os resultados
Pontos ← Ordena(Pontos)
Resultado ← k primeiros arquivos de Pontos
```

1. Etapa 1: O algoritmo busca a primeira palavra na *Lookup Table*.

2. Etapa 2: O algoritmo percorre e descriptografa os elementos do *Array List* referentes a primeira palavra.
3. Estapa 3: O algoritmo realiza as operações de emparelhamento bilinear para cada par arquivo da primeira palavra /outra palavra buscada.
4. Etapa 4: O algoritmo ordena e filtra os arquivos de acordo com as pontuações.

As etapas 1 e 4 (destacadas em vermelho no algoritmo) são executadas de forma sequencial como o algoritmo VRMS tradicional, já as etapas 2 e 3 (destacadas em verde no algoritmo) são executadas em paralelo nos núcleos disponibilizados para a aplicação. A alteração realizada no índice permite que as entradas do *Array List* sejam distribuídas entre os núcleos disponíveis para descriptografar. Desta forma, na etapa 3 as operações de emparelhamento bilinear são divididas entre os núcleos disponíveis. Finalmente, a etapa 4 agrega as pontuações encontradas e determina os arquivos que irão formar o resultado da busca. Esta parte não é paralelizada mas apresenta baixo custo computacional. Note que a divisão das tarefas entre os núcleos nas etapas 2 e 3 não são apresentadas no algoritmo por simplicidade (basta cada núcleo executar um intervalo da busca). Finalmente, a etapa 4 inicia apenas quando todos os núcleos já tenham terminado a computação da etapa 3, esta sincronização é realizada através de mecanismos tradicionais, como barreiras, e também não aparece no algoritmo.

O Algoritmo 1 apresenta em detalhes o *Parallel VMRS*. A entrada é composta pelas 3 estruturas que representam o índice (*Lookup Table*, *Array List* e *QSet*), uma busca com múltiplas palavras $B = \{w_1, w_2, w_3, ..., w_n\}$ e a quantidade máxima de arquivos a ser retornada. Primeiramente, ocorre a inicialização de diversas informações, principalmente os vetores que irão armazenar os dados decodificados do *Array List*. Logo após a inicialização, o *Array List* é decodificado de forma paralela, sendo a quantidade de itens dividida entre os núcleos disponíveis para processamento. Após todos os itens do *Array List* serem decodificados e armazenados, é verificado se para cada par arquivo de w_1 /w_i, ($i = 2, ..., n$) existe uma entrada no *QSet*. Esta operação envolve a requisição de um *token* ao usuário, após o recebimento do *token* é então realiza a exponenciação bilinear. O resultado da exponenciação representa a chave do *QSet*, caso essa chave possua um valor associado, este valor é somado à pontuação do arquivo. A soma da pontuação é realizada em uma seção crítica para evitar condições de corrida. Por fim, o vetor com as pontuações é organizado em ordem decrescente e os primeiros k arquivos são retornados, em que k é a quantidade máxima de arquivos solicitada pelo usuário.

3.3. bVMRS: *Batched* VMRS

O segundo algoritmo proposto, chamado de bVMRS (*Batched* VRMS), aplica tanto paralelismo na execução quanto reaproveitamento de computações dentro de um lote de buscas. O algoritmo evita a execução de operações de exponenciação redundantes através da memorização das exponenciações realizadas nas buscas anteriores. Esta abordagem tem grande potencial de redução no tempo das buscas, pois, por mais que as exponenciações de buscas com diferentes w_1's sejam realizadas com parâmetros diferentes, o resultado representa apenas o par arquivo/palavra, podendo ser reutilizado para qualquer operação que represente este par. Por exemplo, os parâmetros da exponenciação de um arquivo $arquivo\_1$ da busca "casa amarela" com a palavra "amarela" serão diferentes dos parâmetros da exponenciação do $arquivo\_1$ da busca "mesa amarela" com a

palavra “amarela”, porém o resultado é o mesmo, dessa forma é necessária apenas uma operação.

A abordagem utilizada no desenvolvimento do algoritmo bVMRS foi a mesma do pVMRS, em que primeiramente é realizado um pré-processamento. De uma forma geral, o algoritmo funciona em três etapas (Algoritmo 2)

- Etapa 1: Nesta etapa é realizado um pré-processamento, como no algoritmo pVRMS. No entanto, também são decodificados os itens do *Array List* e também é criada uma estrutura com os pares palavra/arquivo únicos que devem ser computados.
- Etapa 2: Nesta etapa é realizado o cálculo de forma paralela entre os núcleos disponíveis. Como os cálculos são realizados considerando cada par definido na etapa anterior, apenas um cálculo é realizado para cada par palavra/arquivo e, na etapa seguinte, o mesmo resultado é reaproveitado para as várias buscas dentro do lote.
- Etapa 3: Nesta etapa ocorre a agregação da pontuação para cada arquivo de cada busca dentro do lote, evitando a seção crítica no laço do cálculo da exponenciação. Por fim, os resultados de cada busca são ordenados e filtrados.

Notar que apenas a etapa 2 é a mais computacionalmente custosa, sendo ainda a única etapa a ser executada de forma paralela. Na primeira etapa os pares palavra/arquivo são definidos, enquanto que na última os resultados das computações são agregados. Desta forma, estas etapas devem ser realizadas de forma sequencial, porém apresentam um custo computacional muito menor quando comparado com a etapa 2.

O Algoritmo 2 apresenta em detalhes o algoritmo proposto. Ao receber um lote de buscas primeiramente ocorre o pré-processamento, em que são decodificados os elementos do *Array List* de cada w_1 de cada busca. Simultaneamente são identificados os pares únicos de palavra/arquivo, e são armazenados em uma tabela *hash* indexada pela tupla cujo valor é a pontuação. Como os itens do *Array List* também armazenam a pontuação relativa ao arquivo e a w_1, esta pontuação também é armazenada na tabela *hash*. Após este pré-processamento, a computação do emparelhamento bilinear é executada em paralelo, onde os itens da tabela *hash* são divididos para execução entre os núcleos disponíveis. Como a estrutura possui apenas pares únicos, esta parte não possui condições de corrida, e portanto não é necessário introduzir uma seção crítica. Por fim os resultados são agregados, ordenados e filtrados, produzindo de uma só vez os resultados para todas as buscas do lote.

Um fator importante e ser considerado é o tamanho do lote a ser processado. Como a busca exige uma comunicação constante com o usuário (para recuperar os *tokens*), é importante que a janela de acumulação das buscas não seja muito longa, evitando que o usuário aguarde tempo demasiado pelo resultado de sua consulta. Dessa forma, esta abordagem adquire melhores resultados para índices com grande quantidade de buscas simultâneas. Um outro fator a ser considerado é a similaridade entre as buscas, quanto mais palavras repetidas houverem entre as buscas, maior é a chance de redução do tempo de busca. Um estudo detalhado a respeito dos ganhos em relação à similaridade é conduzido na seção de avaliação experimental.

Algorithm 2: Algoritmo bVMRS

entrada: Lookup Table lt, ArrayList al, Keyset ks, lote de buscas B, QSet qs e quantidade de arquivos buscados k

saída : Lista de arquivos encontrados para cada busca em B

Etapa 1

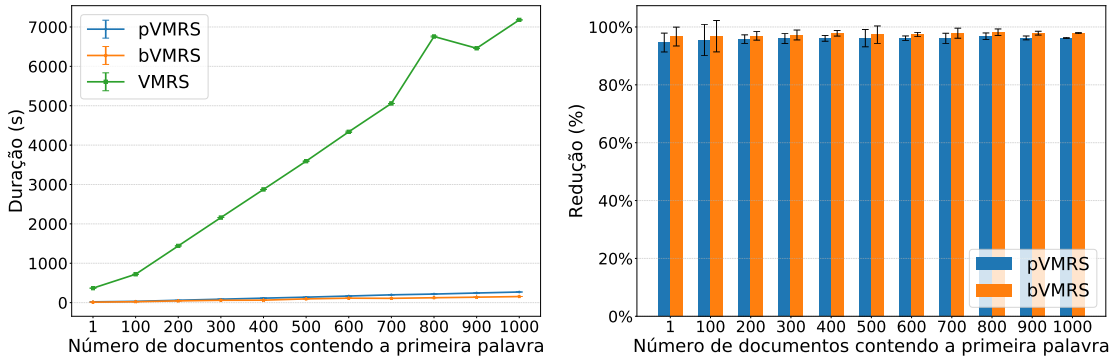
```
// Pré-processamento para definir os pares palavra/arquivo no lote
Pontos ⟵ ∅
Pares ⟵ tabela hash vazia
ParesYic ⟵ tabela hash vazia
Resultado ⟵ matriz de resultados de cada busca
foreach b ← 1 to |B| do
    qtd_outras_palavras ⟵ (quantidade de palavras em b) − 1
    indice_w1 ⟵ lt[b[0]].end
    arqs_w1 ⟵ al[indice_w1]
    for i ← 1 to |arqs_w1| do
        Y_{i,c} ← decodifica de arqs_w1[i]
        Arquivo ← decodifica de arqs_w1[i]
        Pontuacao ← decodifica de arqs_w1[i]
        Resultado[b][i] ← Pontuacao
        Pares[(Arquivo, w1)] ← Pontuacao
        ParesYic[(Arquivo, w1)] ← Y_{i,c}
        for j ← 1 to qtd_outras_palavras do
            if (Arquivo, b[j]) não existe em Pares then
                Pares[(Arquivo, b[j])] ← ∅
                ParesYic[(Arquivo, b[j])] ← Y_{i,c}
            end
        end
    end
end
```

Etapa 2

```
// Recupera a pontuação no QSet uma vez para cada par palavra/arquivo
for i ← 1 to |Pares| do
    token ← SolicitaToken(Pares[i].arquivo, Pares[i].palavra)
    indice_qset ← token^{ParesYic[i].Y_{i,c}}
    if qs[indice_qset] existe then
        Pares[i] ← qs[indice_qset]
    end
end
```

Etapa 3

```
// Agregação, ordenamento e filtro dos resultados
foreach b ← 1 to |B| do
    indice_w1 ⟵ lt[b[0]].end
    arqs_w1 ⟵ al[indice_w1]
    for i ← 1 to |arqs_w1| do
        foreach p em b, exceto a primeira do
            Resultado[b][i] ← Resultado[b][i] + Pares[(arqs_w1[i])]
        end
    end
    Resultado[b] ← Ordena(Resultado[b]) e retorna os k primeiros arquivos
end
```



(a) Tempo médio para realizar 1024 buscas. (b) Porcentagem de redução em relação ao VMRS.

Figura 2. Desempenho do VRMS, pVMRS e bVMRS. pVMRS e bVRMS utilizando 64 *threads* e bVMRS com lotes de 1024.

4. Experimentos

Os algoritmos propostos foram implementados e uma série de experimentos foram realizados para (*i*) comparar os resultados obtidos com o VMRS sequencial [Jiang et al. 2017], possibilitando uma visualização clara dos ganhos relacionados com as soluções propostas, (*ii*) analisar a escalabilidade das soluções propostas, (*iii*) analisar como o tamanho dos lotes afetam o desempenho do bVRMS, e (*iv*) analisar o desempenho do bVRMS considerando diferentes níveis de similaridade entre as buscas.

Implementação. A implementação dos algoritmos propostos foi realizada em linguagem C++, as operações de emparelhamento bilinear foram implementadas utilizando a biblioteca PBC Library [Lynn 2006], a mesma biblioteca utilizada em [Jiang et al. 2017]. As operações criptográficas utilizam a biblioteca CryptoPP [Dai 2004]. O desenvolvimento da paralelização do VMRS foi auxiliado pela API OpenMP [OpenMP ARB 2018], implementada pelo compilador gcc [Gcc 2019]. A comunicação entre usuário e servidor foi abstraída, portanto os resultados não incluem o atraso na comunicação entre as partes.

Ambiente experimental. Os experimentos foram conduzidos em um computador com processador Intel(R) Xeon(R) CPU E5-4620 [Intel Corporation 2012] com frequência de 2.20GHz, 32 núcleos físicos e 64 núcleos virtuais e 12 GB de memória RAM. Além disso, foi utilizado o conjunto de arquivos de e-mail Enron [Cohen 2015]. Assim como em [Jiang et al. 2017], foram escolhidos aleatoriamente 10 mil arquivos e 4 mil palavras para compor o índice.

Resultados e análises. A Figura 2(a) apresenta a duração média de 1024 buscas de acordo com α, em que α é a quantidade de documentos que possuem a primeira palavra da busca. Para cada α foram geradas 1024 buscas com 6 palavras cada. Os resultados apresentados representam a média e desvio de 10 execuções, devido a escala do gráfico não é possível visualizar o desvio. Através deste experimento é possível notar a escala do ganho proveniente da paralelização. O tempo de busca do algoritmo sequencial vai de $367,9s$ a $7179,2s$, enquanto o bVMRS apresenta tempos de $12,2s$ a $154,2s$. A Figura 2(b) apresenta a taxa de redução dos algoritmos paralelos em relação ao VMRS para as mesmas buscas da Figura 2(a). Pode-se notar a drástica redução no tempo de duração alcançada, ficando entre $94,6\%$ e $98,2\%$, uma melhora bastante considerável.

A Figura 3 apresenta a taxa de redução da duração média de lotes de 1024 buscas. As buscas foram escolhidas de forma aleatória, variando seu tamanho de 1 a 10 palavras. O algoritmo bVMRS foi executado utilizando lotes de 1024 buscas. É possível notar uma grande redução aumentando a quantidade de *threads* de 2 para 8, sendo que a partir deste ponto o aumento é menos perceptível. No cenário com a maior quantidade de *threads* utilizadas (64) a redução em comparação ao VMRS é de $96,1\%$ para o pVMRS e $96,1\%$ para bVMRS, indicando que a paralelização apresenta ganhos bastante significativos.

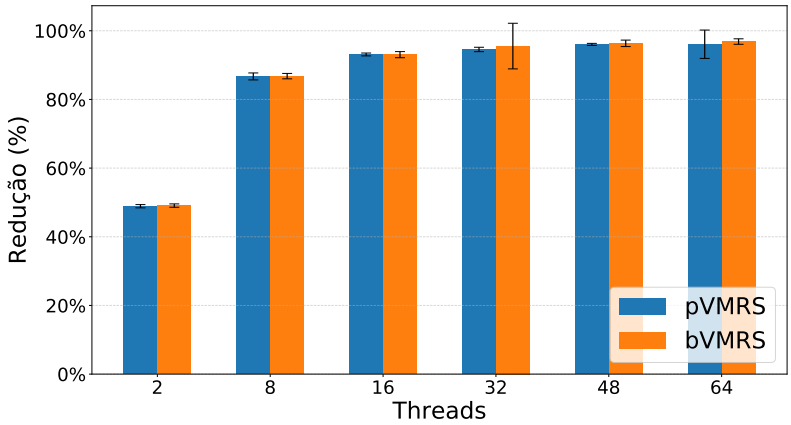


Figura 3. Porcentagem de redução em relação ao VMRS *vs. threads* utilizadas.

A Figura 4 apresenta a taxa de redução do bVMRS em relação ao VMRS e pVMRS, de acordo com o tamanho dos lotes utilizados. Este experimento utiliza as mesmas buscas do experimento anterior. Nesta análise, o ganho do bVMRS em relação ao pVMRS se torna mais evidente. É importante ressaltar que este ganho está diretamente ligado a similaridade entre as buscas de um mesmo lote, e as buscas utilizadas neste experimento foram criadas aleatoriamente sob um dicionário de 4000 palavras, i.e., possuem uma baixa similaridade. Comparando-se com o pVRMS, as taxas de redução variam de 14% a 20%, de acordo com o tamanho do lote. O tamanho do lote é um parâmetro que deve ser configurado de acordo com a quantidade de requisições por unidade de tempo que o servidor recebe, considerando o impacto no tempo de esperada dos usuários. Em relação ao VMRS, pode-se observar um alto ganho mesmo com lotes de 128 buscas.

A Figura 5 apresenta a taxa de redução da duração de 1024 buscas do pVMRS em relação ao bVMRS. Para medir os ganhos do bVMRS foi utilizada uma taxa de similaridade, que representa a porcentagem de pares palavra /documento iguais em um conjunto de buscas, no caso 1024 buscas. Os experimentos demonstram que a estratégia de buscas em lote, utilizada pelo bVMRS, apresenta ganhos inclusive quando não há similaridade entre as buscas, estes ganhos são provenientes da estratégia de implementação, onde primeiramente são separados todos os pares a serem calculados para então o cálculo ser realizado de forma paralela. Também é possível observar que os ganhos, para o lote de 1024 buscas, crescem de acordo com o nível (índice) de similaridade. Os resultados mostram que o bVMRS supera consideravelmente o pVMRS mesmo para níveis baixos de similaridade, atingindo 60% no melhor caso. Sua utilização é preferível quando o servidor recebe uma quantidade considerável de requisições em um curto espaço de tempo.

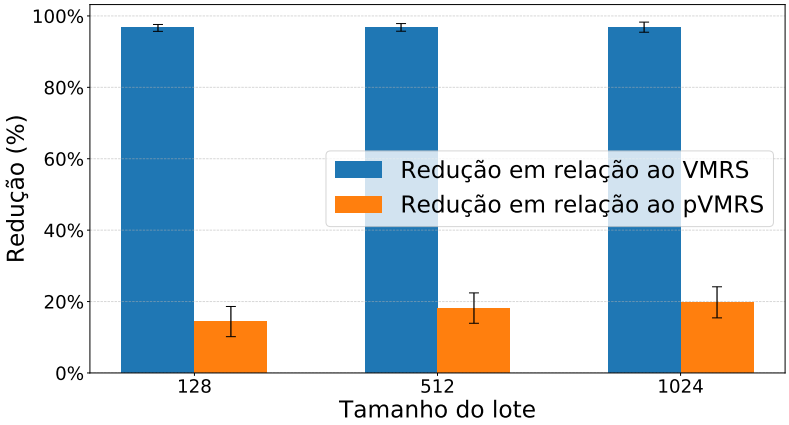


Figura 4. Taxa de redução da duração de 1024 buscas do bVMRS em relação ao pVMRS e ao VMRS.

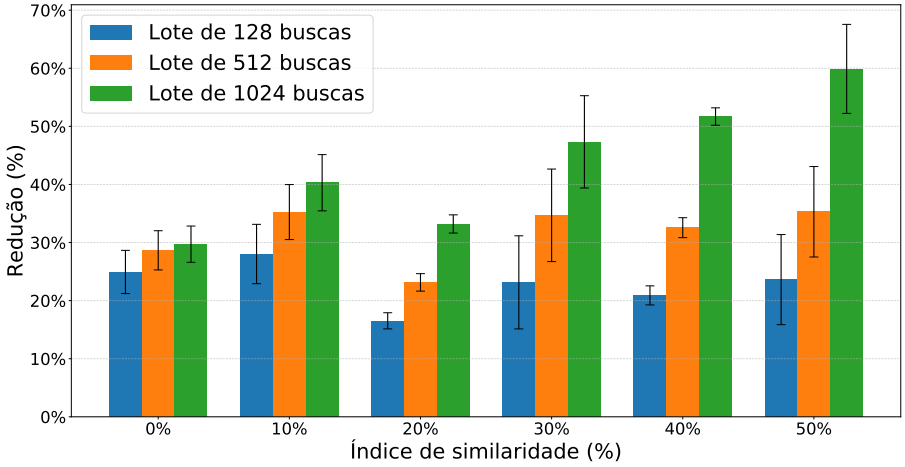


Figura 5. Taxa de redução do bVMRS em relação ao pVMRS de acordo com o índice de similaridade entre as buscas contidas em um lote de buscas.

5. Conclusões

Criptografia buscável é um mecanismo importante no cenário atual de segurança em computação em nuvem, onde é imprescindível garantir a privacidade dos dados dos usuários. Diversas trabalhos apresentaram propostas neste área, permeando funcionalidades e cenários distintos. Este trabalho se baseia em uma proposta que fornece busca de múltiplas palavras, verificável e classificável, e busca melhorar o desempenho das buscas através de paralelismo. Duas abordagens foram propostas e analisadas, o *Parallel VMRS* (pVMRS) e o *Batched VMRS* (bVMRS), o primeiro paraleliza as operações de uma única busca, enquanto que o segundo agrega diversas buscas e utiliza memorização para reduzir a quantidade de operações realizadas. Os algoritmos propostos foram implementados e resultados experimentais demonstram que o pVMRS reduz em mais de 90% o tempo de busca em relação ao algoritmo sequencial, enquanto que o bVMRS apresenta ganhos consideráveis quando comparado com o pVRMS, de acordo com o índice de similaridade entre as buscas, chegando a 60% quando a similaridade entre as buscas atinge 50%.

Referências

Boldyreva, A., Chenette, N., Lee, Y., and O'Neill, A. (2009). Order-preserving symmetric encryption. In Joux, A., editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, Berlin, Heidelberg. Springer Berlin Heidelberg.

Boneh, D. and Franklin, M. (2001). Identity-based encryption from the weil pairing. In Kilian, J., editor, *Advances in Cryptology — CRYPTO 2001*, pages 213–229, Berlin, Heidelberg. Springer Berlin Heidelberg.

Cao, N., Wang, C., Li, M., Ren, K., and Lou, W. (2014). Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):222–233.

Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., and Steiner, M. (2013). Highly-scalable searchable symmetric encryption with support for boolean queries. In Canetti, R. and Garay, J. A., editors, *Advances in Cryptology – CRYPTO 2013*, pages 353–373, Berlin, Heidelberg. Springer Berlin Heidelberg.

Chen, C., Zhu, X., Shen, P., Hu, J., Guo, S., Tari, Z., and Zomaya, A. Y. (2016). An efficient privacy-preserving ranked keyword search method. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):951–963.

Cohen, W. W. (2015). Enron email dataset. `https://www.cs.cmu.edu/~enron/`.

Dai, W. (2004). Crypto++ library a free c++ class library of cryptographic schemes. `http://www.cryptopp.com/`.

Gcc, G. (2019). the gnu compiler collection. `http://gcc.gnu.org`.

Intel Corporation (2012). Intel(r) xeon(r) cpu e5-4620.

Jiang, X., Yu, J., Yan, J., and Hao, R. (2017). Enabling efficient and verifiable multi-keyword ranked search over encrypted cloud data. *Information Sciences*, 403-404:22–41.

Lynn, B. (2006). Pairing-based cryptography library. `https://crypto.stanford.edu/pbc/`.

OpenMP ARB (2018). The openmp api specification for parallel programming. `http://openmp.org`.

Pinkas, B. and Reinman, T. (2010). Oblivious ram revisited. In Rabin, T., editor, *Advances in Cryptology - CRYPTO 2010*, pages 502–519. Springer Berlin Heidelberg.

Poh, G. S., Chin, J.-J., Yau, W.-C., Choo, K.-K. R., and Mohamad, M. S. (2017). Searchable symmetric encryption: Designs and challenges. *ACM Comput. Surv.*, 50(3):40:1–40:37.

Stallings, W., Brown, L., Bauer, M. D., and Bhattacharjee, A. K. (2012). *Computer security: principles and practice*. Pearson Education Upper Saddle River, NJ, USA.

van Dijk, M., Gentry, C., Halevi, S., and Vaikuntanathan, V. (2010). Fully homomorphic encryption over the integers. In Gilbert, H., editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 24–43, Berlin, Heidelberg. Springer Berlin Heidelberg.

Wang, Y., Wang, J., and Chen, X. (2016). Secure searchable encryption: a survey. *Journal of Communications and Information Networks*, 1(4):52–65.