UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Gabriel Luz

Privacidade e verificabilidade de acesso em sistemas médicos usando *searchable encryption*

Florianópolis

2018

Gabriel Luz

Privacidade e verificabilidade de acesso em sistemas médicos usando *searchable encryption*

Monografia submetida ao Programa de Graduação em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Ciências da Computação
Orientador: Prof. Dr. Jean Everson Martina

Florianópolis

2018

Ficha de identificação da obra elaborada pelo autor
através do Programa de Geração Automática da Biblioteca Universitária
da UFSC.

Luz, Gabriel

Privacidade e verificabilidade de acesso em sistemas médicos usando searchable encryption / Gabriel Luz ; orientador, Jean Everson Martina, 2018.
125 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2018.

Inclui referências.

1. Ciências da Computação. 2. Sistemas médicos. 3. Searchable encryption. 4. Privacidade. 5. Verificabilidade. I. Martina, Jean Everson. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Privacidade e verificabilidade de acesso em sistemas médicos usando searchable encryption.

Gabriel Luz

Privacidade e verificabilidade de acesso em sistemas médicos usando *searchable encryption*

Esta Monografia foi julgada adequada para obtenção do Título de "Bacharel em Ciências da Computação" e aprovada em sua forma final pelo Programa de Graduação em Ciências da Computação

Florianópolis, 28 de junho de 2018.

_____
Prof. Dr. Rafael Luiz Cancian
Coordenador do Curso

Banca Examinadora:

_____
Prof. Dr. Jean Everson Martina
Orientador

_____
Ma. Thaís Bardini Idalino

_____
Ma. Dayana Pierina Brustolin Spagnuelo

Dedico este trabalho à minha namorada, por estar sempre ao meu lado, me apoiando e ajudando, em especial nos momentos mais difíceis ao longo desta jornada. Também dedico à minha irmã pelo seu apoio e à minha mãe, que sempre lutou para dar uma boa educação aos seus dois filhos. Sem estas pessoas, este trabalho não seria possível.

AGRADECIMENTOS

Agradeço ao Professor Dr. Jean Everson Martina pela orientação e compreensão ao longo deste trabalho e pelo auxílio dado para superar os obstáculos no caminho. Agradeço também a participação da banca formada por Thaís Bardini Idalino e Dayana Pierina Brustolin Spagnuelo, as quais tornaram este trabalho possível, por meio de seus estudos anteriores sobre o assunto e do auxílio durante o desenvolvimento do mesmo.

RESUMO

Sistemas médicos têm se tornado populares pois facilitam o compartilhamento de dados entre pacientes e médicos. Entretanto, o compartilhamento deve estar sob controle do paciente, que irá decidir que informações devem ser compartilhadas e quem deve ter acesso a elas. O paciente deve ter o poder de verificar os acessos que foram feitos a seus dados, podendo assim confirmar se houve algum acesso que não esteja de acordo as regras estabelecidas. O paciente pode não ter o conhecimento técnico necessário para fazer esta verificação. Logo, é desejável também que o paciente possa enviar os registros dos acessos para um verificador, juntamente com suas preferências de acesso, para que possa ser verificado se houve alguma quebra das regras de acesso definidas pelo paciente. Dessa maneira, o processo de verificação seria facilitado pelo verificador, aumentando as chances dos pacientes utilizarem e realizarem essa verificação frequentemente. Entretanto, o uso de um verificador pode comprometer a privacidade dos dados pois é possível obter diversas informações somente com os registros de acesso, como qual médico foi visitado ou quais exames foram realizados. Portanto, ao permitir que outra pessoa acesse estes registros, o sigilo das informações do paciente pode estar comprometido. Soluções no estado da arte utilizam técnicas de *searchable encryption* para solucionar este problema. Este trabalho visa implementar um modelo proposto no estado da arte em um protótipo de sistema médico e avaliar a aplicabilidade do modelo.

Palavras-chave: Sistemas médicos. *Searchable Encryption*. Banco de Dados. Verificabilidade. Privacidade.

ABSTRACT

Medical systems have risen in popularity because they make the data sharing between patients and the medical staff easier. However, the patient must be in control of this data sharing. The patient must be able to verify the access to his data to confirm if there has been any violation to his privacy preferences. It is desirable that the patient can send the log entries to his data and his data sharing preferences to a third-party that could verify if there has been any breach to his privacy. However, it is possible to extract a lot of personal information given the access logs to someone's medical data such as medical exams and doctor visits. Some solutions on the state of the art use searchable encryption schemes to solve this issue. In this work, one of the proposed solutions was be implemented along with a prototype of a medical system using this solution to simulate a real life scenario and verify its applicability.

Keywords: Medical systems. Searchable Encryption. Databases. Verifiability. Privacy.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API – Application Programming Interface……
FHIR – Fast Healthcare Interoperability Resources……..
JSON – JavaScript Object Notation……...
XML – Extensible Markup Language……
REST – Representational State Transfer……....
DNF – Disjunctive Normal Form……...
RPC – Remote Procedure Call……...

SUMÁRIO

“I love deadlines. I love the whooshing noise they make as they go by.”
— Douglas Adams, The Salmon of Doubt

1 INTRODUÇÃO

Quando o programa de espionagem dos Estados Unidos veio a conhecimento público pelas denúncias realizadas por Edward Snowden, o tópico da privacidade de dados tornou-se extremamente popular (BBC NEWS, 2016). A variedade de dados contendo informações pessoais tem crescido exponencialmente nas últimas décadas (SWEENEY, 2002). Apesar de existir uma grande discussão no campo da privacidade sobre o que deve ser considerado sigiloso e o que deve ser aberto ao público, ainda é um consenso que dados médicos de pacientes são dados sigilosos, que só poderão ser compartilhados sob o consentimento do paciente. Existem várias soluções apresentadas no estado da arte para resolver problemas relacionados à privacidade e verificabilidade usando *searchable encryption*.

Searchable encryption é um tipo de criptografia que permite que terceiros realizem uma consulta no dado cifrado, satisfazendo uma condição sem ter que decifrar o texto. Para isso, alguns esquemas de criptografia consultável requerem que se tenha uma palavra-chave para ser usada como um *search token* para cada consulta, vinculando uma palavra-chave para cada dado para que seja possível fazer uma pesquisa somente com as palavras-chave (*search tokens)*. Logo, para esquemas que usam *search tokens*, para cada consulta é necessária a criação de um *search token* por quem possuir a chave e o envio do *token* para quem realizará a operação de consulta.

A verificação de acesso é comumente requerida em uma vasta área de tecnologias, principalmente quando se trata do acesso a dados médicos, onde o sigilo não pode ser quebrado. Aplicações que requerem a verificabilidade encontram problemas para garantir a confiabilidade dos dados e também para dar a sensação de segurança aos usuários de que seus dados estão sendo armazenados com sigilo.

O modelo proposto por Idalino, Spagnuelo e Martina (2017) usa conceitos de verificabilidade e *searchable encryption* para solucionar problemas relacionados à verificabilidade de acessos em sistemas médicos. O *framework* proposto possibilita que o paciente verifique se houve algum acesso não autorizado aos seus dados e também possibilita que este autorize um terceiro a fazer essa verificação, enviando os registros de acesso a essa pessoa. Este processo

é feito sem que esta pessoa que fará a verificação consiga obter qualquer informação a partir dos registros de acesso.

1.1 JUSTIFICATIVA

O modelo utilizado propõe uma solução para o problema de verificabilidade da privacidade dos dados de pacientes em sistemas médicos. Entretanto, trata-se de um estudo inicial que ainda não possui uma implementação que comprove sua aplicabilidade. Uma implementação do modelo proposto contribui para o estado da arte provendo a possibilidade de uma prova de conceito deste trabalho e uma análise da aplicabilidade do mesmo.

1.2 OBJETIVOS

1.2.1 Objetivo geral

Implementar um modelo proposto para a solução do problema de verificabilidade em sistemas médicos. Após a implementação, avaliar a aplicabilidade do projeto.

1.2.2 Objetivos específicos

- Implementar o modelo proposto por Idalino, Spagnuelo e Martina (2017) para a verificação de acesso a sistemas médicos.

- Implementar um protótipo de sistema médico com acesso via portal web pelo paciente e acesso a um verificador para verificar os logs de acesso.

- Avaliar a aplicabilidade prática do modelo com base no protótipo desenvolvido neste trabalho. Levando em consideração a funcionalidade do protótipo e os testes realizados.

1.3 METODOLOGIA

Para atingir os objetivos descritos na seção anterior, a metodologia desenvolvida divide o trabalho em três fases.

A primeira fase tem como principal objetivo uma pesquisa no estado da arte para compreender amplamente o problema. Para isso, será realizada uma pesquisa de casos de uso de sistemas médicos e uma revisão bibliográfica de criptografia consultável e suas aplicações. Além disso, será pesquisado algumas implementações de sistemas médicos para uma melhor contextualização, avaliando dessa forma a relevância da implementação deste trabalho em um cenário aplicado.

Na segunda fase, serão decididas as principais definições para a realização deste trabalho. Primeiramente, será escolhido um algoritmo de *searchable encryption* que suporte a *queries* booleanas para a implementação deste trabalho. Em seguida, serão definidos quais os principais módulos a serem implementados. Com isso, será decidido quais tecnologias serão usadas, levando em consideração a afinidade com a tecnologia, o desempenho para uma maior relevância do trabalho e a contextualização do trabalho.

Na terceira e última fase, serão implementados os objetivos deste trabalho com base nas definições realizadas nas fases anteriores. O primeiro passo da implementação é a criação de uma biblioteca de *searchable encryption* para ser usada pelos próximos módulos. Com a biblioteca pronta e testada, o segundo passo é realizar testes de consulta com *searchable encryption* e desenvolver uma maneira de aplicar as necessidades da expressividade de busca booleana do trabalho de Idalino, Spagnuelo e Martina (2017) utilizando a biblioteca. O terceiro passo é a criação dos logs a serem usados neste trabalho. O quarto e último passo é a criação do Sistema Médico, Verificador e do Cliente que utilizam a biblioteca de *searchable encryption* desenvolvida. Após a implementação, será discutida a aplicabilidade deste trabalho, levando em consideração o protótipo desenvolvido neste trabalho.

1.4 ORGANIZAÇÃO DOS CAPÍTULOS

Os capítulos deste trabalho serão apresentados da seguinte forma:

- Fundamentação teórica: Apresentação dos conceitos julgados necessários para o entendimento do conteúdo a ser discutido nesse trabalho.

- Modelo: Explicação do modelo a ser usado para implementação, suas suposições e resultados esperados.

- Implementação: Organização do projeto de implementação, componentes usados, trechos de código considerados cruciais para o trabalho.

- Conclusão: Apresentação do que foi obtido com este trabalho, sua importância e contribuição para o estado da arte em privacidade de dados. Além disso, será feita uma discussão a respeito de possíveis áreas a serem exploradas, as quais não entraram no escopo deste trabalho.

“Sometimes life's going to hit you in the head with a brick. Don't lose faith. I'm convinced that the only thing that kept me going was that I loved what I did.”
— Steve Jobs

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão introduzidos conceitos avaliados necessários para o entendimento do trabalho realizado.

2.1 CRIPTOGRAFIA

Criptografia é o estudo de técnicas matemáticas relacionadas à segurança da informação, que engloba itens como confidencialidade, integridade, autenticação de dados e da origem (MENEZES; VAN OORSCHIT; VANSTONE, 1996). Cifrar é uma técnica de transformação da informação de uma forma legível para uma forma ilegível utilizando uma chave. A informação ilegível só pode ser transformada em algo legível novamente com a posse de uma chave. O processo de transformação de informação legível em ilegível, transformando informação em *plaintext (*texto plano) para *ciphertext* (texto cifrado) é chamado de cifragem.

A criptografia é um dos grandes pilares do mundo moderno, desde o seu uso limitado pelos Egípcios há 4000 anos atrás e começando com seu emprego crucial durante as grandes guerras no século XX (MENEZES; VAN OORSCHIT; VANSTONE, 1996). Hoje, a criptografia não é somente usada para proteger segredos nacionais, mas também para proteger a privacidade de cada cidadão.

A importância da criptografia no mundo moderno começou depois do uso de computadores para armazenamento de informações, sendo usada para proteger os dados digitais. Desde então, várias técnicas de criptografia foram propostas para resolver diversos problemas.

Para exemplificar, será utilizado um cenário onde Alice e Bob desejam trocar mensagens privadas entre si e Eve deseja ver ou alterar o conteúdo dessas mensagens. A Figura 1 mostra um cenário de comunicação insegura entre Alice e Bob com interceptação da mensagem por Eve. Comunicação insegura é uma comunicação onde se assume que alguém como Eve possa interceptar e alterar mensagens.

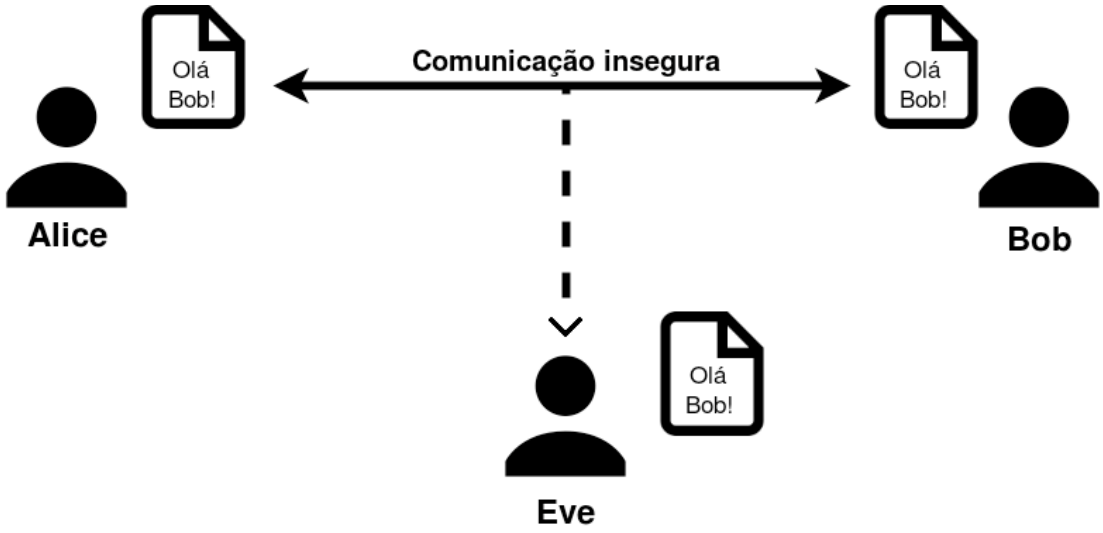

Figura 1 - Troca de mensagens insegura

2.1.1 Criptografia Simétrica

A criptografia simétrica é uma classe de criptografia que utiliza somente uma chave. Esta chave consegue cifrar e decifrar a mensagem. No exemplo da Alice e do Bob, é preciso que eles definam uma chave com uma comunicação segura e então poderão trocar mensagens seguras em um canal não seguro. Quem tiver posse dessa chave consegue decifrar as mensagens trocadas. Este cenário descrito pode ser visto na Figura 2 onde Alice e Bob possuem a mesma Chave c e Alice manda a mensagem “Olá Bob” cifrada com a Chave C para Bob que então pode decifrar com a mesma Chave c. Já que Eve não possui a Chave c, não consegue decifrar o conteúdo da mensagem.

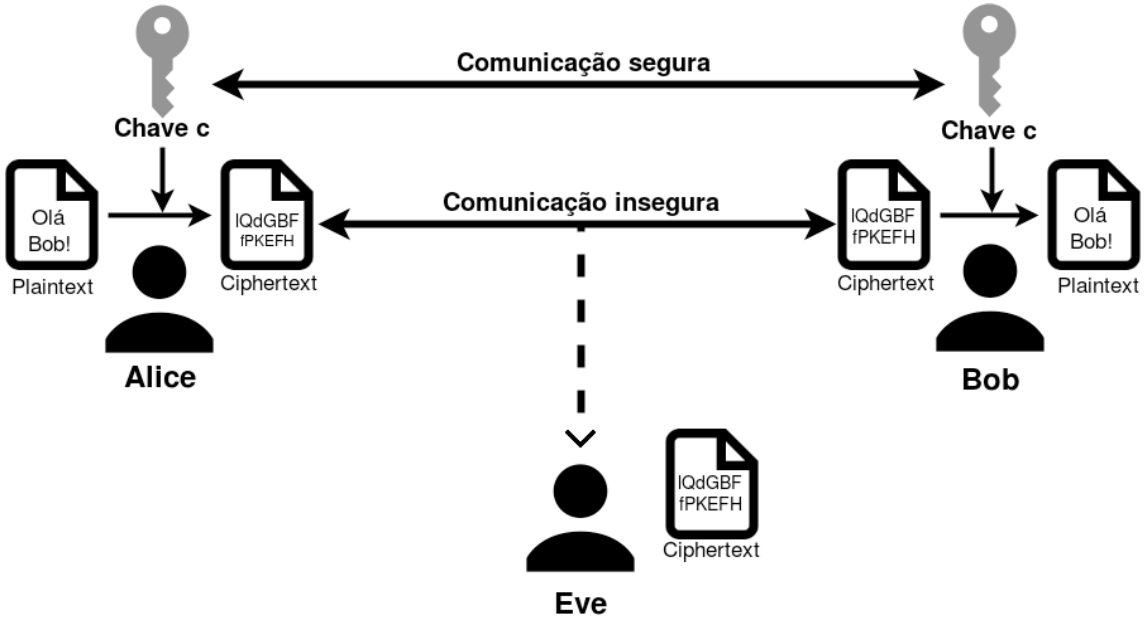

Figura 2 - Troca de mensagens com criptografia simétrica

Uma das dificuldades da utilização da criptografia simétrica é encontrar um meio seguro para o acordo de chave. Esse tema foi ponto de partida para vários trabalhos, um deles foi o algoritmo de Diffie-Hellman que torna possível duas pessoas combinarem uma chave por um canal inseguro (DIFFIE; HELLMAN, 1976). Uma alternativa é não utilizar chave simétrica, eliminando a necessidade de um acordo de chaves é o uso de criptografia assimétrica descrito na próxima seção.

2.1.2 Criptografia Assimétrica

A criptografia assimétrica, também conhecida como criptografia de chave pública, é outra classe de criptografia a qual utiliza um par de chaves diferentes. Neste tipo de criptografia, uma chave é pública, enquanto a outra é secreta. O texto cifrado que uma chave produz só pode ser decifrado com a outra chave. A Figura 3 mostra um cenário da Alice e do Bob onde ambos disponibilizariam sua chave pública, para que a Alice possa mandar uma mensagem privada para Bob. A Alice deve cifrar a mensagem com a chave pública do Bob, e então somente o Bob terá a chave privada que pode decifrar essa mensagem.

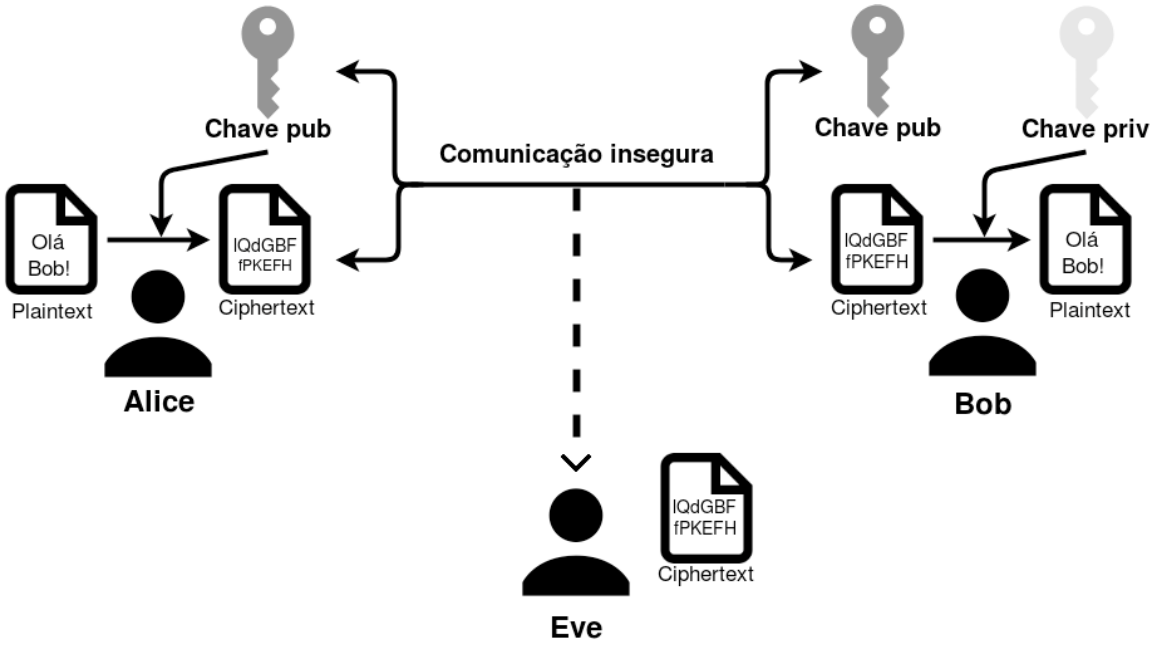

Figura 3 - Troca de mensagens com criptografia assimétrica

2.2 SEARCHABLE ENCRYPTION

O termo utilizado na literatura é *Searchable Encryption*, porém, o autor pode se referenciar a *searchable encryption* como criptografia consultável como tradução direta ao longo do texto. A criptografia consultável permite que um servidor faça buscas em documentos sem que possa extrair qualquer informação destes documentos (ABDELRAHEEM, 2016).

Alguns esquemas de criptografia consultável possibilitam fazer pesquisas no *ciphertext* enquanto a maioria dos esquemas permite a geração de um *index* com *keywords* que são utilizadas para a geração da *query* e a pesquisa posteriormente (BÖSCH et al., 2015). Para pesquisar, o cliente gera uma *query* com os predicados que deseja verificar e envia para o servidor, que por sua vez itera sobre os *index* e retorna os documentos que satisfazem o predicado. Os tipos de criptografia consultável e suas vantagens e desvantagens serão discutidas na seção 4.2 na escolha e implementação da criptografia consultável para este trabalho.

A Figura 4 ilustra um cenário *client-server* utilizando *searchable encryption.* O primeiro passo é o cliente gerar os índices cifrados para serem mandados à cloud junto com os dados cifrados.

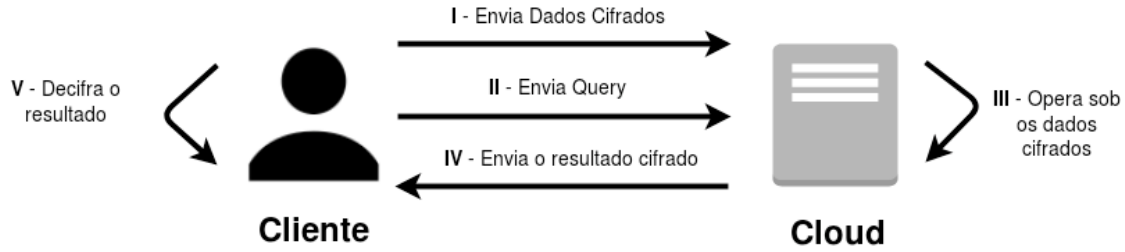

Figura 4 - *Searchable Encryption* em um cenário *client-cloud*

Os algoritmos de criptografia consultável clássicos só permitem a consulta de uma única *keyword,* porém muitos problemas, como o apresentado neste trabalho, necessitam de um algoritmo mais expressivo onde é possível fazer *queries* booleanas. Por exemplo, com um algoritmo clássico é possível fazer a query dos logs de dados médicos onde a *ob* é igual a 'X-RAY'. Porém, no escopo deste trabalho é necessário o uso de *queries* mais expressivas como por exemplo, uma consulta onde *ob* é igual a 'X-RAY' e *actor* é igual a '1'.

De acordo com Kamara e Moataz (2017) uma solução ingênua de realizar essa consulta denominada de *response-revealing* seria fazer a pesquisa onde *ob* é igual a 'X-RAY', outra pesquisa onde id é igual a '1' e realizar a intersecção entre os dois resultados. O problema desta solução ingênua é que apesar de não vazar diretamente as *keywords*, o *server* teria os índices da primeira e da segunda consulta. Alguns tipos de ataques poderiam descobrir informações sensíveis sobre os dados cifrados com um número significante de *queries*. Com um algoritmo que suporta *queries* mais expressivas o servidor teria somente os índices da intersecção, ou seja, o equivalente a uma pesquisa de uma única *keyword*.

Outra solução ingênua denominada de r*esponse-hiding* por Kamara e Moataz (2017) são esquemas que o *server* não consegue descobrir os índices que são somente calculados no cliente, o problema de soluções como esta, de acordo com os autores, são que são ineficientes por passar informações redundantes. No caso do exemplo anterior, o problema de informações redundantes seria que o servidor responderia todos os índices onde *ob* é igual a *'X-RAY'* e todos os índices onde *actor* é igual a '1' e o cliente resolveria a intersecção.

Ainda de acordo com Kamara e Moataz (2017), qualquer esquema de criptografia consultável deve resolver um desses dois problemas das soluções ingênuas.

2.3 DADO, INFORMAÇÃO E CONHECIMENTO

Os termos dado, informação e conhecimento normalmente são usados como sinônimos fora de um contexto formal. Porém, é importante definir as diferenças entre os três termos para um compreendimento melhor ao longo deste trabalho. Os dados são informações não tratadas, não podem transmitir mensagem ou conhecimento. Eles são a matéria prima para a informação. As informações são os dados tratados, utilizam os dados para adicionar contexto e agregação de valores aos dados e podem ser usados para tomar decisões ou afirmações. O conhecimento pode ser visto como uma informação tratada e trabalhada, do mesmo jeito em que a informação é vista como um dado trabalhado. O conhecimento é usado por uma pessoa para criar, armazenar e compartilhar informações.

2.4 CONFIDENCIALIDADE, INTEGRIDADE, DISPONIBILIDADE, AUTENTICIDADE E IRRETRATABILIDADE

Confidencialidade, integridade, disponibilidade, autenticidade e irretratabilidade são consideradas as propriedades básicas da segurança da informação e são importante para um compreendimento completo do trabalho desenvolvido na área de segurança da informação. A confidencialidade consiste em que somente pessoas autorizadas terão acesso ao dado decifrado. No contexto deste trabalho, os registros de acesso do paciente devem garantir a propriedade de confidencialidade, onde somente o paciente e o sistema médico consigam acessar os registros. A integridade garante que a informação mantenha todas as suas características originais ao longo do seu ciclo de vida. A disponibilidade garante que a informação esteja sempre disponível. A autenticidade garante que a fonte anunciada pela informação seja verídica. A irretratabilidade ou não repúdio, garante que não seja possível negar a autoria.

2.5 RESUMO CRIPTOGRÁFICO

Resumo criptográfico, também conhecido como *hash*, é uma função matemática onde é extremamente difícil computacionalmente descobrir o valor de entrada dado o valor de saída. O valor de saída normalmente é chamado de resumo criptográfico ou *hash*. Deve ser difícil encontrar duas mensagens com o mesmo *hash,* assim como modificar uma mensagem mantendo o mesmo *hash*. Quando isso acontece, o algoritmo é considerado quebrado e deixa de ser usado.

2.6 ASSINATURA DIGITAL

Com a criptografia de chave pública foi possível criar a assinatura digital, um método que permite as propriedades de integridade, autenticidade e não repúdio. Ela utiliza funções de resumo criptográficos e criptografia de chave pública.

A Figura 5 ilustra o ato de assinar um documento. Para assinar, primeiramente o assinante cria um hash do documento usando funções de resumo criptográfico. Logo em seguida, ele utiliza a sua chave privada cifrar o *hash*. Este resultado é a assinatura digital, que deve ser enviado juntamente com o algoritmo de cifragem e o documento para verificação da assinatura.

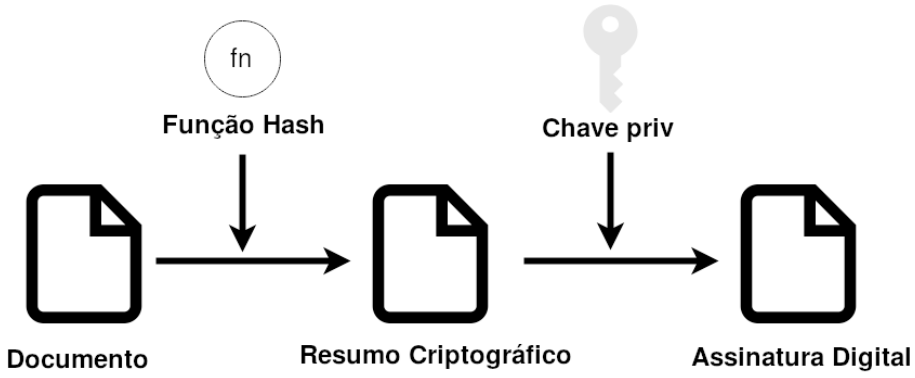

Figura 5 - Ato de assinar com assinatura digital

A Figura 6 ilustra o processo de verificação de assinatura digital. O receptor recebe o documento junto com a assinatura digital e

para verificar a assinatura digital, ele precisa fazer o caminho reverso do processo de assinatura. O receptor aplica a chave pública de quem supostamente assinou o documento na assinatura para obter o hash. Em seguida, ele a função de resumo criptográfico no documento e é comparado este *hash* com o resultado anterior. Se forem iguais, isso prova a integridade do documento pois ambos os *hashes* são iguais confirmando que são resultados da função de resumo criptográfico do mesmo documento. Também prova a autenticidade da assinatura e o não repúdio pois somente o assinante pode realizar esta assinatura com sua chave privada.

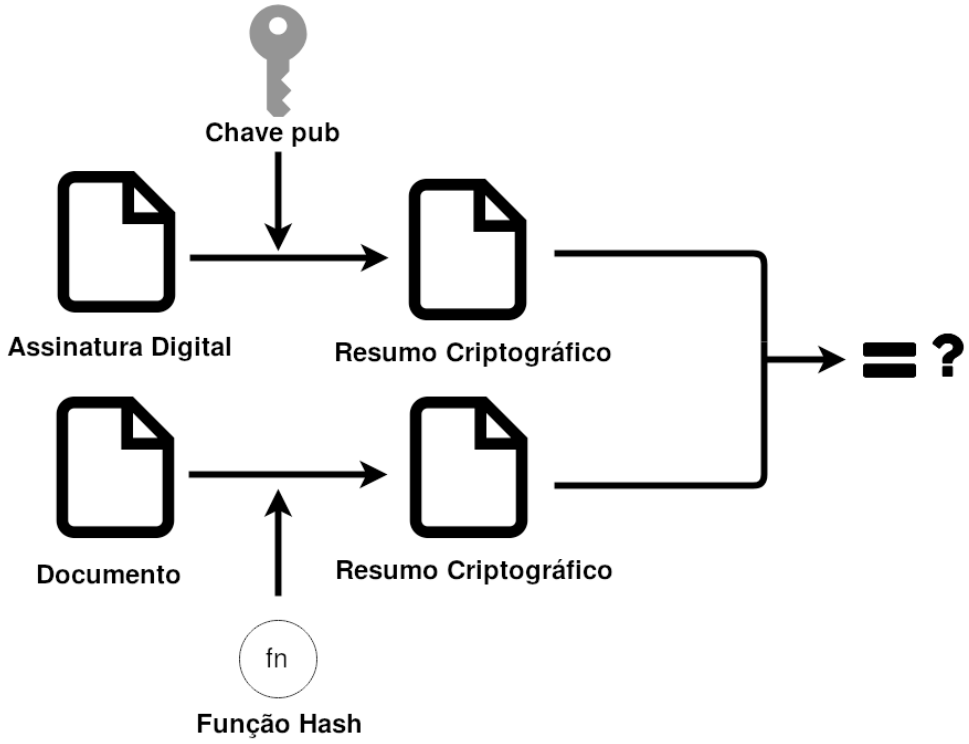

Figura 6 - Verificação de assinatura digital

2.7 CERTIFICADO DIGITAL

O certificado digital é um documento digital que contém a chave pública junto com outras informações importantes da entidade qual possui a chave privada. As informações contém o tipo de algoritmo, nome do titular, a chave pública, o período de validade entre outras informações. O órgão emissor do certificado é quem atesta a veracidade de que o titular possui a chave privada correspondente a esta chave pública.

2.8 SISTEMAS MÉDICOS

Sistemas médicos é um termo genérico que abrange diferentes tipos de sistemas com diversas funcionalidades e objetivos. Uma meta em comum entre todos eles é a coleta e armazenamento de informações da saúde de um paciente em um formato digital. Essas informações podem ser disponibilizadas para o paciente ou a equipe médica. Essas informações incluem dados como histórico médico, dados demográficos, medicamentos, alergias, testes de laboratório, imagens de radiografia entre outras informações (KIERKEGAARD, 2011). O sistema também pode conter informações adicionadas pelo paciente como por exemplo seu peso por meio de uma balança digital conectada a seu celular e outras informações manualmente adicionadas pelo paciente como humor ou dor. A grande vantagem de um sistema médico é o poder de busca que ele oferece para a equipe de saúde e a segurança de manter os dados armazenados digitalmente. Sem um sistema médico, o hospital tem que manter uma organização física de informações relacionadas a seus pacientes, o paciente também deve manter uma cópia de seus exames e quando realizar uma visita ao hospital deve procurar todos os exames relevantes e levar consigo. Em muitos casos esses exames podem ser perdidos ou o paciente pode não levar exames relevantes para a visita. O uso de sistemas médicos soluciona esses problemas de perda e replicação de dados e possibilita a equipe médica encontrar tendências no histórico unificado do paciente. Além disso, se o sistema médico for amplamente adotado, principalmente em instituições públicas atingindo uma grande parte da população, possibilita o estudo estatístico muito preciso (TANG et. al. 2006).

Este trabalho terá foco em sistemas médicos orientados aos pacientes. São sistemas onde o paciente tem poder sobre seus dados. As definições de um sistema de saúde com essa filosofia começaram a aparecer no começo dos anos 2000 (MARKLE FOUNDATION, 2003). No começo eram definidos sistemas onde o paciente mantém as informações no sistema atualizadas, adicionando seus resultados de exame, diagnósticos, informações geradas pelo paciente entre outras informações mencionadas anteriormente. Além de armazenar as informações, algumas implementações do sistema também possibilita a troca de mensagens entre o paciente e o médico, marcar consultas e

exames e lembretes. Eliminando essas barreiras de comunicação e possibilitando a troca de informações facilmente, sistemas médicos poupam o tempo gasto com consultas e podem salvar vidas auxiliando a equipe médica a diagnosticar problemas de saúde precocemente ou receitar um tratamento eficiente para a saúde do paciente baseado nos resultados dos tratamentos anteriores.

Antes do surgimento de sistemas médicos existiam outras maneiras de manter um sistema por meio de diferentes plataformas que conseguiam atingir algumas das funcionalidades dos sistemas médicos atuais. A mais básica é manter arquivos impressos em papéis e ter uma organização de arquivos, para compartilhar com o médico tirar cópias dos documentos necessários e levar para a visita. O armazenamento em papel consegue atingir a funcionalidade de organização dos dados como um sistema médico, porém depende do usuário o nível de organização e tem problemas com perda de dados em caso de algum desastre, ou esquecimento do usuário em levar outros exames necessários. Além dessa plataforma, era comum manter o sistema médico com dispositivos eletrônicos, porém localmente. Pode ser utilizado um *software* específico de sistemas médicos, onde o usuário mantém uma cópia dos dados local que podem estar cifrados ou não, e o usuário deve combinar um meio de comunicação para compartilhar os dados com o seu médico. O problema com este meio é a falta de um padrão para compartilhamento com os funcionários da saúde e o usuário ser responsável pelos *backups* de seus dados. Por último, soluções modernas que gerenciam o compartilhamento, recuperação de dados, adição de informações automática. Esse tipo de plataforma promove os pontos positivos anteriormente citados, porém gera algumas questões controversas. Uma das questões é como essa tecnologia pode ameaçar a privacidade dos usuários. O medo dos usuários em ter todos os seus dados médicos online pode atrasar a adoção dos sistemas médicos. Soluções que promovem a garantia de confiabilidade e maior segurança para os pacientes em relação aos registros de acesso, como a apresentada neste trabalho, podem transformar a expressividade da adoção dos sistemas médicos.

2.8.1 Protocolo FHIR

Fast Healthcare Interoperability Resources, ou FHIR, é um protocolo criado para definir formato de dados e uma API para a troca de dados entre sistemas médicos (BENDER; SARTIPI, 2013). O protocolo foi criado por uma organização sem fins lucrativos chamada *Health Level Seven International* envolvida no desenvolvimento de padrões para sistemas médicos.

Esse protocolo foi criado a partir de outros protocolos já utilizados para troca de mensagens de sistemas médicos como as antigas versões do HL7 (BENDER; SARTIPI, 2013), um protocolo também criado pela mesma organização . O objetivo do *FHIR* é trazer um padrão mais fácil de implementar, utilizando tecnologias modernas ainda mantendo interoperabilidade com sistemas antigos. Os dados podem ser representados em formato *JSON* ou *XML*, e a comunicação é feita utilizando uma *API RESTful*.

2.8.2 Implementações de sistemas médicos

Várias implementações de sistemas médicos foram criadas, as mais notáveis são das maiores empresas de tecnologia no mundo como a *Google*, *Microsoft* e *Apple*. A *Google* apostou em sistemas médicos com o lançamento do produto *Google Health* em 2008 (LOHR, 2011), porém foi finalizado em 2011 por falta de adoção de usuários. O serviço possibilitava os pacientes a inserirem seus dados médicos manualmente ou automaticamente se forem dados em empresas parceiras. O motivo acreditado ser o principal para a não adoção do *Google Health* foi a desconfiança dos usuários em fornecer seus dados para a empresa. Farzard Montashari, antigo coordenador da tecnologia e informação da saúde no departamento de saúde e serviços humanos dos Estados Unidos, acredita que o *Google Health* estava a frente do seu tempo e que desde então tem sido criado *standards* para compartilhamento de dados médicos (CNBC, 2017). A *Microsoft* também lançou seu produto em sistemas médicos chamado de *Microsoft HealthVault* lançado em 2007 e oferece também os mesmos serviços esperados de um sistema médico (THE ECONOMIST, 2007). Cada indivíduo tem o seu *Vault* que pode compartilhar com outras pessoas. Após a saída da *Google* deste mercado o *HealthVault* recebeu um fluxo grande de usuários e permaneceu sozinha por um tempo no mercado, inclusive diminuindo as equipes de pesquisa e desenvolvimento. Em 2016 a *Apple* lançou o

Apple CareKit que é um *SDK* que permite o desenvolvimento padronizado e rápido de sistemas médicos que permitiu o desenvolvimento de diferentes sistemas médicos. O *Apple CareKit* é uma aposta diferente de mercado do *HealthVault*, pois não é um sistema desenvolvido somente pela *Apple*, são ferramentas que podem ser utilizadas para o desenvolvimento padronizado de sistemas médicos na plataforma da *Apple*.

2.8.2.1 Apple Health

A *Apple* lançou em 2015 o *framework* chamado *ResearchKit*, ela possibilitava pesquisas médicas em massas utilizando o *iPhone* (EARL, 2015). Com a adoção rápida do *ResearchKit* e a qualidade de seus resultados, fazendo pesquisas médicas com milhões de pessoas somente utilizando seus celulares, perceberam que poderiam ampliar muito mais a área da saúde nos seus aparelhos (UMER, 2016). Então, lançaram o *framework* chamado *CareKit*. *CareKit* é um *framework open source* que auxilia provedores de saúde a criarem aplicativos que dão poder aos usuários. Eles permitem o acompanhamento muito mais pessoal entre o paciente e o médico. Um exemplo seria o acompanhamento pós-operatório onde normalmente é dado um papel com instruções iguais para todo o paciente e elas são estáticas, ou seja, não mudam independente da recuperação do paciente. O *CareKit* possibilita que o paciente responda perguntas de como está sua recuperação, dando escala para dor, e suas instruções pós-operatórias são dinâmicas, mudando de acordo com a recuperação do paciente.

Quando lançaram o *CareKit*, cada implementador criava seu esquema de comunicação de dados entre o paciente e o hospital, assim como a privacidade de acesso. A *Apple* percebeu que o compartilhamento de dados entre o paciente e o sistema médico era uma área crítica, então em 2017 lançaram uma *API* pronta para a comunicação entre o sistema médico e o aplicativo utilizando a framework CareKit (DEKOSHKA, 2017).

Porém, o CareKit ainda é simplesmente um *framework* para implementar sistemas médicos, então depende de cada sistema médico para implementar as questões de privacidade com os dados dos usuários.

“War is peace.
Freedom is slavery.
Ignorance is strength.”
― George Orwell, 1984

3 MODELO

O modelo proposto por Idalino, Spagnuelo e Martina (2017) tem como objetivo promover o poder aos usuários de sistemas médicos de verificar a concordância dos acessos ao sistema médico com suas preferências de acesso à seus dados médicos. Além disso, o usuário também tem o poder de verificar com um terceiro, onde o verificador não poderá extrair informações a partir dos logs de acesso. Cada paciente tem sua política de privacidade, contendo o que cada membro da equipe médico pode acessar de seus dados.

Os autores assumem três diferentes participantes nesse sistema, são eles:

- Sistema médico: Armazena os dados médico dos pacientes e é responsável por compartilhar os dados entre os participantes e fornecer uma lista de logs com os acessos aos dados do paciente
- Paciente: Usuário do sistema, dono das informações contidas no sistema médico. Pode pedir acesso aos logs do sistema médico para verificar por si mesmo ou enviar para qualquer terceiros para fazerem a verificação.
- Verificador: Terceiro que pode fornecer serviços para verificação de acesso a dados de pacientes sem conseguir obter informações sobre o paciente.

Além disso, os autores também definem requerimentos para o sistema como:

- Verificação Automatizada: O sistema médico deve fornecer meios para facilitar a verificação para o paciente.
- Auditoria de dados independente: Um terceiro pode verificar os dados de acesso aos dados do paciente.
- Privacidade: Em nenhum momento a privacidade do paciente deve ser violada.

Para isso, foi utilizado criptografia consultável, que permite fazer buscas no dado cifrado sem revelar informações sobre os dados. Os logs de registro de acesso precisam passar pelo algoritmo de criptografia consultável utilizado, gerando os índices e as *keywords* de cada índice. O paciente e o sistema médico fazem um acordo de uma chave simétrica, que é usada para cifrar os índices e as *keywords*. O

paciente, por sua vez, com posse da chave em que foi cifrado os índices e as *keywords*, deve criar uma *query* de acordo com as suas preferências de acesso. Então, ele pode enviar os dados cifrados para o verificador junto com a *query*, que pode auditar esses logs cifrados e retornar o resultado afirmando se todos os logs respeitam as preferências de acesso ou não, sem obter alguma informação sobre o paciente. A Figura 7 ilustra o modelo proposto.

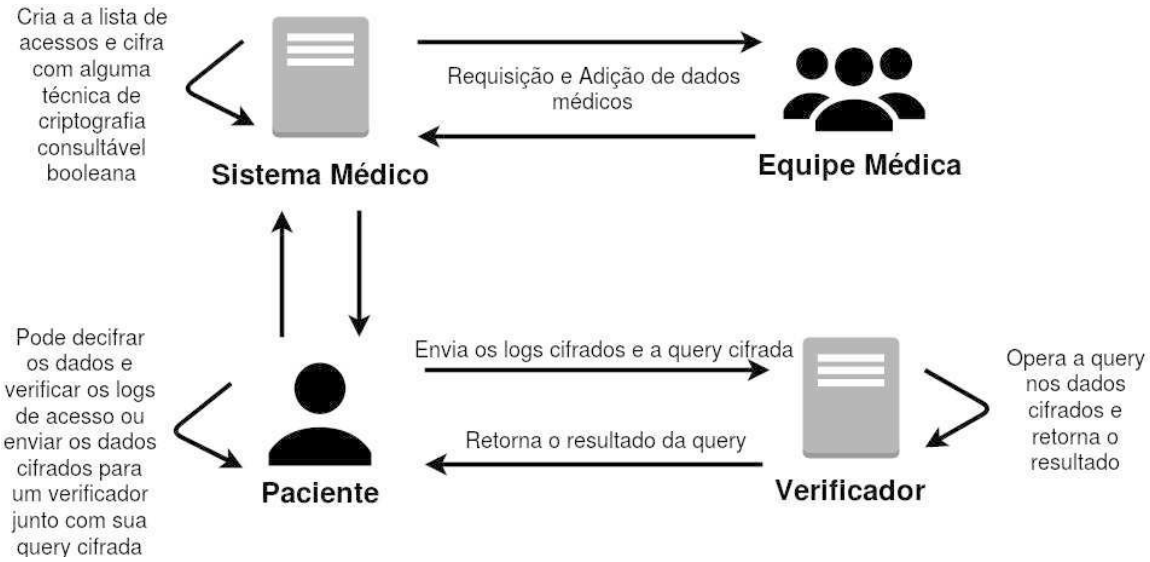

Figura 7 - Modelo

O modelo deles assume o sistema médico como honesto mas não confiável. Desse modo, o sistema não é malicioso, mas pode ser abusado, como alguém acessar algum dado médico que não poderia. Eles consideram o verificador honesto mas curioso, se tiver alguma informação sobre o paciente disponível ele não respeitaria o sigilo. Eles também consideram a existência de um atacante que pode reter qualquer informação disponível ao verificador.

Para evitar colusão entre o sistema médico e o verificador, os autores recomendam a implementação de verificadores por diferentes entidades. Dessa forma, colusões seriam evitadas, pois a possibilidade de um paciente utilizar outro verificador honesto é muito alta e o sistema médico perderia completamente a sua credibilidade, o que é o objetivo oposto da utilização deste trabalho. Além disso, verificadores podem ser verificados enviando uma requisição de verificação onde o resultado é conhecido e comparando com a resposta do verificador.

O trabalho proposto pelos autores não leva em consideração a integridade dos logs de acesso por isso ser de responsabilidade do sistema médico. Eles assumem que o sistema médico mantém logs seguros e íntegros.

3.1 ALGORITMOS UTILIZADOS PELO MODELO

O algoritmo descrito na Figura 8 é o qual o sistema médico gera os logs de acesso cifrados. O sistema médico cifra cada log individualmente, criando um *index* para cada um, contendo as palavras-chave do log, como a ação e o autor dessa ação. A geração do *index* depende do algoritmo de criptografia consultável usado.

Algorithm 1 EncryptLogs($logs[n]$)

Input: array of $logs$ with n entries
for each $i \in \{1, \ldots, n\}$ **do**
    $c[i] = \text{Enc}(k, logs[i])$
    $\text{keywords} = \text{extractKeywords}(logs[i])$
    $\text{index}[i] = \text{generateIndex}(k, \text{keywords})$
end for
Output: c, index

Figura 8 - Algoritmo EncryptLogs retirado de Idalino, Spagnuelo e Martina (2017)

Depois de gerar e cifrar os índices com as keywords de cada log, eles são enviados ao paciente. Ele, por sua vez, pode escolher analisar os logs por sí mesmo ou dar essa tarefa a um verificador. Caso decida enviar para um verificador, é necessário explicar ao verificador o que ele deve buscar nesses logs. Para isso, o paciente gera uma query. A Figura 9 mostra esse processo da geração da query. A query é gerada de acordo com a política de privacidade acordada com o sistema médico. Essa política de privacidade define o que cada membro da equipe médica pode acessar nos dados do paciente. Se a política de privacidade não for alterada, o paciente pode reutilizar essa query futuramente. A query é representada como uma expressão booleana na disjunctive

normal form (DNF), que é uma normalização de uma disjunção de cláusulas conjuntivas, utilizando um id identificando a pessoa e um action identificando a ação que essa pessoa pode realizar. Uma query ficaria no seguinte formato (id1 /\ action1) \/ (id1 /\ action2) \/ (id2 /\ action1). É utilizado a DNF pois com ela é possível realizar a disjunção de cláusulas conjuntivas, nesse caso é a disjunção das conjunções dos ids dos membros médicos com os actions que eles podem realizar. Logo, é uma query que verifica se um log está de acordo com a política de privacidade.

Algorithm 2 GenerateQuery(π)

Input: Policy $\pi = \{(id_i, action_i)\}$ with m clauses
DNF = empty string
for each $i \in \{1, \ldots, m\}$ **do**
    DNF = DNF $\|(id_i \wedge action_i)$
    if $i \neq m$ **then**
        DNF = DNF $\|\vee$
    end if
end for
$\mathcal{Q}$ = generateQuery(DNF)
Output: $\mathcal{Q}$

Figura 9 - Algoritmo GenerateQuery retirado de Idalino, Spagnuelo e Martina (2017)

Depois de gerar a query, o paciente deve enviar os logs cifrados e a query para o verificador. Isso é suficiente para o verificador fazer a busca nos logs e retornar os que estão de acordo com a query. A busca por parte do verificador é mostrada na Figura 10. Note que este algoritmo também depende do esquema de criptografia consultável booleana utilizado. Depois de realizar a verificação, o verificador envia para o paciente uma lista de logs cifrados que estão de acordo com a *query*.

Algorithm 3 Search($\mathcal{Q}, c[n], index[n]$)

Input: Encrypted query $\mathcal{Q}$, vector c with n encrypted logs, and their indexes
for each $i \in \{1, \ldots, n\}$ **do**
  $r = \text{test}(\mathcal{Q}, index[i])$
  if $r =$ **true then**
    $result$.add($c[i]$)
  end if
end for
Output: $result$

Figura 10 - Algoritmo Search retirado de Idalino, Spagnuelo e Martina (2017)

O resultado pode ser simplificado mandando uma mensagem positiva ou negativa se há violação da política de acesso do sistema médico a partir do resultado da *query* a partir da condição de que todos os índices estão de acordo com a *query*. O algoritmo 2 de geração da query também pode ser facilmente modificado para indicar quais logs violam a política e o paciente pode decifrar esses logs para descobrir qual foi a violação. Em nenhum momento o verificador obtém qualquer dado sobre os logs de acesso, ou sobre a *query*. A única informação para ele é a quantidade de *logs* que estão de acordo com a *query* enviada, que não tem seus termos em texto plano para o verificador.

“It is not enough that we do our best; sometimes we must do what is required.”
― Winston S. Churchill

4 IMPLEMENTAÇÃO

A implementação tem diferentes componentes, são eles:

- Uma pequena biblioteca de criptografia consultável chamada *OpenSSE*.
- O sistema médico: Armazena os dados dos pacientes e fornece os logs de acesso cifrados.
- Um verificador: Opera com os logs de acesso cifrados dos pacientes.
- O sistema para o paciente que pode criar sua query cifrada e verificar o resultado.

A implementação do sistema médico, verificador e paciente foram feitas em classes distintas que implementam o protocolo *JSON-RPC* junto com suas respectivas aplicações web que se comunicam via *JSON-RPC*. Para o uso da técnica de criptografia consultável foi criada uma pequena biblioteca nomeada pelo autor deste trabalho de *OpenSSE*, para que possa ser utilizada pelas classes do sistema médico, verificador e paciente.

Com estas classes e a biblioteca desenvolvidas, foram feitos os testes e experimentos envolvendo as três diferentes classes para analisar a aplicabilidade do projeto.

4.1 TECNOLOGIAS UTILIZADAS

Para a implementação do sistema médico e verificador foi decidido utilizar a linguagem C++. Foi utilizado o protocolo de comunicação *JSON-RPC* para a comunicação com as funções desenvolvidas. Para a chamada das aplicações web para suas respectivas funções via *JSON-RPC* foi utilizado *jQuery*. Para o armazenamento da política de acesso, será utilizado o formato *JSON*.

Para implementar as chamadas JSON-RPC foi utilizado a biblioteca *libjson-rpc-cpp* que permite a criação de classes que implementam o protocolo *JSON-RPC* a partir de um arquivo de especificação das chamadas RPC no formato JSON. Com isso, é

possível definir todas as chamadas RPC de uma classe. Um exemplo de arquivo de especificação e descrito abaixo:

```
[
        {
                "name": "sayHello",
                "params": {
                        "name": "Peter"
                },
                "returns" : "Hello Peter"
        },
        {
                "name" : "notifyServer"
        }
]
```

Com o arquivo *JSON* de especificação, é possível gerar as classes abstratas de *Server* e *Client* com o comando:

```
jsonrpcstub spec.json --cpp-server=AbstractStubServer --cpp-client=StubClient
```

Este comando gera as classes abstratas de *Server* e *Client.* Basta implementar os métodos abstratos definidos nas especificações em uma classe herdando o *AbstractStubServer* e iniciar o servidor que qualquer client *JSON-RPC 2.0 compliant* pode se comunicar. Para o *Client* basta herdar da classe *StubClient* iniciar um objeto *StubClient* e chamar os métodos RPC pelo objeto.

4.2 IMPLEMENTAÇÃO DE UM ALGORITMO DE CRIPTOGRAFIA CONSULTÁVEL BOOLEANA

Para a implementação do algoritmo de criptografia consultável foi escolhido o algoritmo proposto por Abdelraheem et al. (2016). Os autores propuseram um algoritmo de criptografia consultável que permite operações booleanas. Apesar de que este algoritmo não foi diretamente mencionado no trabalho proposto por Idalino, Spagnuelo e Martina (2017), este trabalho é uma melhoria do trabalho proposto por

Cash et al. (2013) citado pelos autores. Ele possui as características necessárias apontadas pelos autores para ser utilizado como o algoritmo de criptografia consultável neste modelo. A mais importante sendo a de utilização de query com múltiplas *keywords*. Além disso, Abdelraheem et al. (2016) disponibilizou os código dos testes realizados em sua pesquisa, o que facilitou a implementação da biblioteca *OpenSSE*.

Com isso, a implementação da criptografia consultável foi a adaptação do código de algoritmo de criptografia consultável disponibilizado por Abdelraheem et al. (2016) em uma pequena biblioteca chamada *OpenSSE.* Ela possui duas classes *Client* e *Server*. O *Client* é responsável por coletar as *keywords* e *indexes* e cifrar utilizando uma chave escolhida. O *Client* também é responsável pela criação da *query* e verificação do resultado. O *Server* realiza a consulta nos *indexes* e *keywords* cifrados.

O código disponibilizado por Abdelraheem et al. (2016) foi feito para realizar os testes de desempenho disponibilizados em seu trabalho, as funções do *Client* e *Server* se comunicavam entre si via *sockets*. Com isso, a primeira alteração necessária foi abstrair os algoritmos de criptografia consultável para que funcionassem independente da conexão do *Client* com o *Server.*

No trabalho desenvolvido, por utilizar um número relativamente pequeno de dados, não tem problema com a complexidade dos algoritmos, somente com a confidencialidade. Os algoritmos de criptografia consultável propostos por Cash et al. (2013) e Abdelraheem et al. (2016) são soluções baseadas em *response-hiding*. A *queries* nestes esquemas são feitas na forma *w1 ^ f(w2, w3, ..., wn)*. Os autores definem dois grupos, *s-tag* e *x-tags*. O grupo *s-tag* seria o *w1* e os elementos do grupo das x-tags seriam as *keywords w2*,*w3*,*wn*. Neste algoritmo, o servidor realiza uma consulta de única palavra com o *w1* e obtém os índices, a partir disso ele faz uma consulta com cada *x-tag* somente nos índices obtidos pelo *w1*. O *server* envia para o *client* os índices das *x-tags* cifrados, ou seja, o único conhecimento do server é a da *query* de *w1*, como uma consulta de palavra única. Após receber os resultados, o *client* pode decifrar os índices das x-tags e realizar as operações booleanas, no caso do trabalho dos autores seria *w1 ^ w2 ^ w3 ^ wn.* O problema de informações redundantes neste algoritmo proposto é feito pela conjunção lógica entre a *s-tag* e as *x-tags*. Para obter um melhor desempenho teria que se escolher uma *s-tag* com a

menor frequência para economizar operações para as *x-tags.* A figura 11 ilustra a interação *Client* <-> *Server.*

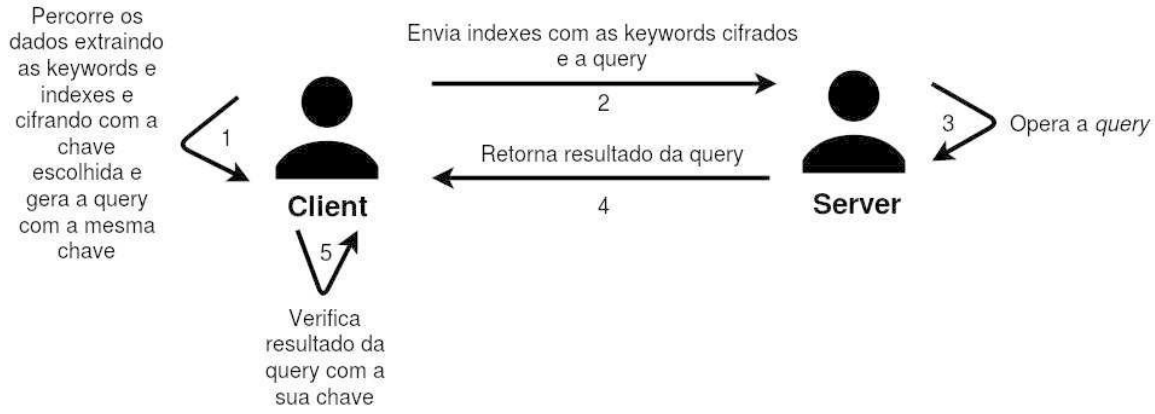

Figura 11 - Interação Client *Server* na OpenSSE

Este trabalho visava utilizar a criptografia consultável para realizar consultas no formato ((*w1 AND w2) OR (w3 AND w4) OR (wn AND wn))* onde cada conjunção é entre o *id* e *action*. Dessa forma, não é possível utilizar o mesmo formato apresentado no trabalho de Abdelraheem et al. (2016). Para isso, foi adaptado para que o resultado das *x-tags* seja resolvido depois de ter chamado a classe *Client* para resolver o resultado onde é possível realizar qualquer operação booleana entre as *x-tags*. Com esta utilização de esquema de criptografia consultável com *response-hiding*, como alguns dos propostos por Idalino, Spagnuelo e Martina (2017), é necessário que o paciente opere sobre o resultado do verificador, como pode ser visto no novo modelo na Figura 12. Já que a *query* para cada log não é composta por múltiplos termos, e sim a possibilidade de fazer mais de uma pesquisa de única palavra sem vazamento de informações, precisa ser alterada a forma de procurar por violações, sem o uso da *DNF*, esse novo algoritmo será apresentado na implementação do sistema do Paciente.

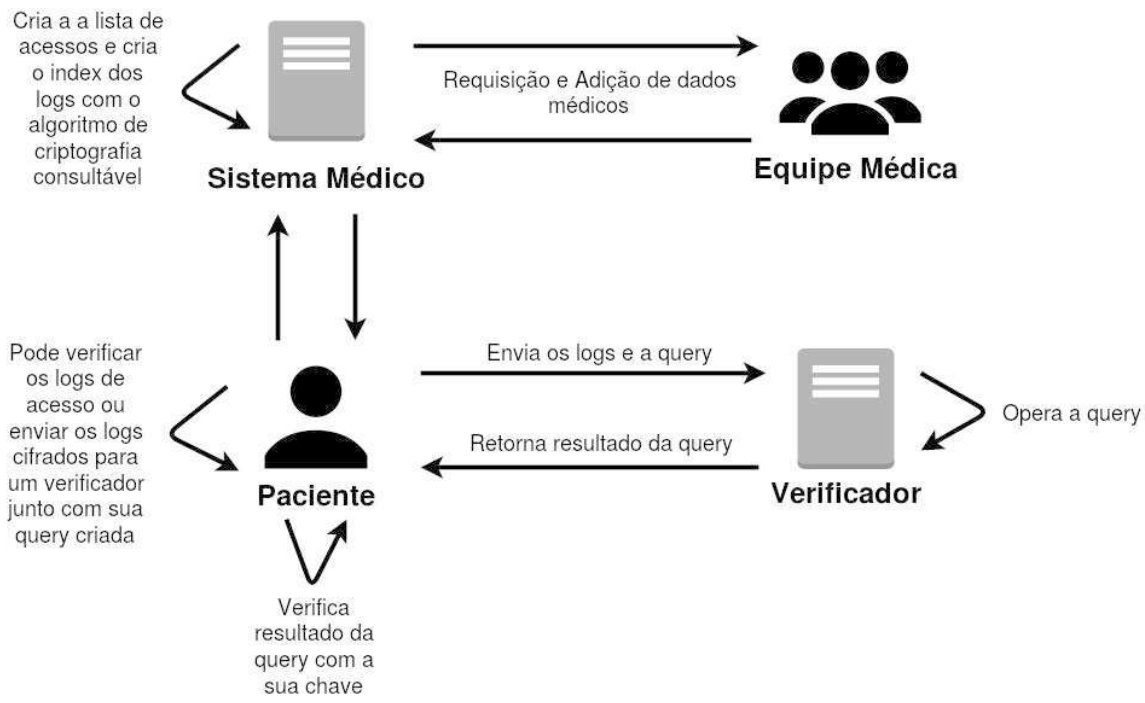

Figura 12 - Modelo adaptado para criptografia consultável *response-hiding*

A Classe *Client* possui 4 métodos importantes, são eles: *setupDB, singleSearch, conjunctiveSearch* e *fetchResult*. O método *setupDB* realiza a leitura dos arquivos e cria a lista de *indexes* e *keywords* análogo ao algoritmo descrito na Figura 8. Este método recebe como parâmetro a localização dos arquivos, o limite de arquivos para leitura e onde ele deve escrever o arquivo de saída dos *indexes* e *keywords* cifrados. Para facilitar a implementação, todas as funções utilizam uma chave que está *hard-coded*. O método *setupDB* antes das alterações realizadas, abria um socket para mandar as informações dos *indexes* e *keywords* para o cliente.

O método *conjunctiveSearch* da classe *Client* cria a query para busca com múltiplas *keywords*, ela recebe como parâmetro a *s-tag* e as *x-tags*. Com a chave que está *hardcoded* ela cifra a *query* e retorna a *query cifrada.* Antes das alterações realizadas, este método abria um *socket* com o *Server* e mandava a *query cifrada*.

O método *fetchResult* da classe *Client* retorna um vetor de *bitsets* contendo os *indexes* que contém a *x-tag* a partir do resultado do *Server.* Esse método recebe como parâmetro o resultado da operação da

query conjunctiveSearch no *Server*. Com isso, ela decifra cada resultado das *x-tags* e adiciona o *bitset* contendo os índices que contém a *x-tag* a um vetor. Antes das alterações, o método *fetchResult* era um método privado da classe, chamado no método *conjunctiveSearch* para decifrar o resultado e então realizar operações lógicas *AND* com todas as *x-tags*.

A classe *Server* possui três métodos importantes, são eles: *setupEDB, singleKeywordSearch e conjunctiveSearch*. O método *setupEDB* foi criado para que o objeto da classe *Server* inicializar suas *s-tag* e *x-tags* para as buscas a partir de um arquivo, ele recebe como parâmetro a localização do arquivo contendo os *indexes* e *keywords* cifrados. Após executar o *setupEDB* com um objeto da classe Server é possível chamar os métodos *singleKeywordSearch* e *conjunctiveSearch.* O método *conjunctiveSearch* recebe como parâmetro a *query* e a localização para o arquivo de saída. Ela primeiro executa uma *singleKeywordSearch* com a s-tag, e então para cada *index* que contém essa *s-tag* ele executa a query de cada *x-tag* e escreve o resultado no arquivo de saída.

4.3 LOGS DE ACESSO

Os logs de acesso estão em um único arquivo. Eles são organizados na seguinte forma:

ACTION:GET DATE:2018-05-14 ACTOR:1 OB:X-RAY

Neste formato, o campo *ACTION* pode ser *GET* ou *SUBMIT*, representam o ato de pegar informações ou incluir informações sobre o paciente respectivamente. O campo *DATE* representa a data que foi realizada esta ação. O campo *ACTOR* representa o ator dessa ação, no caso alguém da equipe médica identificado pelo seu *id* numérico. O campo *OB* representa o artefato médico submisso a esta ação.

Para simplificar a implementação, todos os logs gerados para o trabalho são de um único paciente. Os logs são iterados pela função *setupDB* do *Client* na biblioteca *OpenSSE*.

Foram criados 10 *ids* para o campo *ACTOR* que representam 10 participantes da equipe médica. Para o campo *OB* foram utilizados

alguns termos para dados de sistemas médicos. São eles: *X-RAY, URINE-SAMPLE, DNA, MRI, LDL, HDL, VLDL, VHS, T3, T4*. Totalizando 10 dados médicos, em sua maioria são resultados de exames de sangue para entrar no contexto de sistemas médicos.

4.4 POLÍTICA DE ACESSO DO USUÁRIO

As preferências de acesso dos dados médicos dos usuários são definidas por um arquivo JSON que define a regra para cada tipo de dado médico com um array de ids que podem acessar estas informações de acordo com as preferências do usuário. As preferências de acesso usadas foram a descrita abaixo.

```
{
   "patient": "John Doe",
   "allow": {
            "X-RAY": [1, 2, 3, 4],
            "URINE-SAMPLE": [2, 3],
            "DNA": [9],
            "MRI": [7],
            "LDL": [6],
            "HDL": [6],
            "VLDL": [6],
            "VHS": [1],
            "T3": [1,4],
            "T4": [1,4]
   }
}
```

Desta maneira, o paciente “John Doe” permite, por exemplo, que os médicos com id de 1 e 4 podem acessar os resultados do exame de sangue *t3* e *t4*. O paciente permite que várias pessoas acessem dados do *X-RAY* como os médicos com *ids 1, 2, 3, 4*.

4.5 SISTEMA MÉDICO

O Sistema Médico mantém os logs de acesso e as preferências dos usuários. Ele tem duas funções, são elas: *GetPatientPreferences, GenerateEncryptedLogs*. No escopo deste protótipo, o *GetPatientPrefernece*s retorna o JSON relacionado ao paciente *John Doe*. Para gerar os *indexes* e *keywords* cifrados, o sistema médico cria um objeto *Server* da biblioteca *OpenSSE* e pede para cifrar para um arquivo. O fornecimento dos logs pelo sistema médico está fora do escopo deste protótipo.

```
[
   {
      "name": "GetPatientPreferences",
      "returns" : {}
   },
   {
      "name": "GenerateEncryptedLogs",
      "params": {
         "output_path": ""
      },
      "returns" : false
   }
]
```

A função *GenerateEncryptedLogs*, no caso deste protótipo, precisa iniciar o *Client* da biblioteca *OpenSSE* e chamar a função *setupDB* com os parâmetros sendo: a localização dos logs utilizados, o número máximo de logs a serem percorridos e o caminho para o arquivo de saída. A chamada *JSON-RPC* retorna *true* caso tenha finalizado corretamente. A biblioteca OpenSSE possui a função de geração das chaves, para o escopo dos testes deste projeto foi gerado uma chave e utilizada essa mesma já gerada para os testes.

```
bool MedicalSystemServer::GenerateEncryptedLogs(const std::string&
output_path)
{
   Client *c = new Client();
   c->setupDB("/home/luz/tcc/libssebitmap/resources/datasets/logs",
30, output_path);
```

```
    return true;
}
```

A função *GetPatientPreferences* simplesmente retorna as preferências estabelecidas para John Doe neste experimento.

```
Json::Value MedicalSystemServer::GetPatientPreferences()
{
    Json::Value return_value;
    std::ifstream
file("/home/luz/tcc/libssebitmap/medsystem/preferences.json");
    file >> return_value;
    return return_value;
}
```

4.6 PACIENTE

O software usado pelo paciente possui uma chamada para geração da *query* pela chamada *JSON-RPC GenerateQuery* que recebe o arquivo *JSON* de preferências e retorna uma *string* com a *query*. A outra chamada *JSON-RPC* necessária pelo paciente é a da verificação dos resultados chamada *CheckResult*, que precisa como parâmetro as preferências do usuário no formato *JSON* e retorna uma *string* se houve ou não violação de acesso aos logs.

```
[
    {
        "name": "GenerateQuery",
        "params": {
            "preferences": {}
        },
        "returns" : ""
    },
    {
        "name": "CheckResult",
        "params": {
            "preferences": {},
```

```
            "result_path": ""
        },
        "returns" : ""
    }
]
```

A geração dos logs cifrados do sistema médico descrita na seção 4.6 gera todos os logs, incluindo os que enviam dados (*SUBMIT*) e acessam dados (*GET*). Por isso, como *s-tag* foi escolhido ser sempre a *ACTION == GET* já que a lógica entre as preferências de acesso do paciente serão trabalhadas em cima das *x-tags* e o *JSON* de preferências. Com essa *s-tag* o servidor operará somente nos *indexes* que acessam dados. Se for necessário realizar buscas com *ACTION* em um valor diferente de GET teria que ser feita uma outra *query* com a respectiva *s-tag*

Para a geração das *x-tags* na *query*, foi criado a seguinte organização, para cada OB, adicionar os *ACTORs* que podem acessar em seguida. Sendo OB o objeto, como por exemplo *X-RAY,* e os *ACTORs* seguidos são os ACTOR*s* que podem acessar esse *OB*.

No exemplo dado pelas preferências de *John Doe* a *query* ficaria da seguinte maneira:

OB:X-RAY ACTOR:1 ACTOR:2 ACTOR:3 ACTOR:4 OB:URINE-SAMPLE ACTOR:2 ACTOR:3 OB:DNA ACTOR:9 OB:MRI ACTOR:7 OB:LDL ACTOR:6 OB:HDL ACTOR:6 OB:VLDL ACTOR:6 OB:VHS ACTOR:1 OB:T3 ACTOR:1 ACTOR:4 OB:T4 ACTOR:1 ACTOR:4

Para isso, o método *GenerateQuery* precisa receber as preferências do usuário no formato *Json*. Abaixo está a função *GenerateQuery.* Para cada membro do *array allow* ele adiciona a string das *x-tags* a *x-tag* "*OB:* membro" e para cada *ID* permitido desse OB ele adiciona a *x-tag* ID: id, respeitando o formato *OB, IDs permitidos*

```
std::string PatientServer::GenerateQuery(const Json::Value& preferences)
    {
```

```
        Client *c = new Client();
        std::string stag("ACTION:GET");
        std::string xtags = "";

        Json::Value allow = preferences["allow"];
        std::vector<std::string> keys = allow.getMemberNames();
        for(auto iter = keys.begin(); iter != keys.end(); ++iter) {
           xtags += "OB:" + (*iter) + " ";
                        for (Json::Value::ArrayIndex i = 0; i != allow[(*iter)].size(); i++) {
              xtags+= "ACTOR:" + allow[(*iter)][i].asString() + " ";
           }
        }
        std::string query = c->conjunctiveSearch(stag, xtags);
        return query;
     }
```

Com a *string* da *query* criada, é instanciado um objeto *Client* da biblioteca *OpenSSE* e chamada a função para criação da *query conjunctiveSearch* com base nessa string que retorna uma *string* com a *query* cifrada que pode ser enviada para o *Server*. Já que para o escopo do protótipo deste trabalho foram utilizadas uma chave previamente gerada, o *Client* já conhece o caminho dela.

A outra chamada *JSON-RPC* necessária pelo Paciente é a de verificação de resultado. Com essa organização das *x-tags* essa função executa a mesma lógica para verificar as condições de acesso de cada *x-tag*. Para isso, ele precisa chamar o objeto do *Client* da biblioteca *OpenSSE* para decifrar o resultado. Como o caminho de onde está o resultado do verificador é conhecido nestes testes ele está diretamente no código. Essa função também necessita da *chave* previamente utilizada, como explicado anteriormente, ela já é conhecida nesse protótipo, então o caminho dela está diretamente no código.

```
c->fetchResult(result_path);
```

O *fetchResult* retorna um vetor de dynamic bitsets que representam os resultados de cada *x-tag*. Esses bitsets são compostos

por *0's* e *1's* que representam os *índices* que contém essa *x-tag*. Para verificar o resultado, o algoritmo percorre as preferências do usuário junto com o vetor de *strings* obtido pelo *fetchResult,* para cada resultado da *x-tag* do tipo *ACTOR* faz a operação booleana *OR* com um bitset temporário, depois é feito a operação booleana *AND* com o resultado do *OB* e então a negação desse resultado são os índices dos logs onde o acesso foi de um *OB* mas foi acessado por um *ACTOR* diferente do especificado na política de acesso, é feita a operação OR com um bitset de violações para o retorno. No final, ele verifica se houve algum índice que viola a política de acesso do usuário checando o bitset *violations* e retorna uma *string* com o resultado. A função está descrita abaixo.

```
std::string PatientServer::CheckResult(const Json::Value&
preferences, const std::string& result_path)
    {
        Client *c = new Client();
        std::vector<boost::dynamic_bitset<unsigned char>> result;
        result = c->fetchResult(result_path);

        //Algorithm to check the results
        Json::Value allow = preferences["allow"];
        std::vector<std::string> keys = allow.getMemberNames();
        auto result_iter = result.begin();
        boost::dynamic_bitset<unsigned char> violations
((*result_iter).size());
        for(auto iter = keys.begin(); iter != keys.end(); ++iter) {
            boost::dynamic_bitset<unsigned char> ob;
            ob = (*result_iter);
            boost::dynamic_bitset<unsigned char> actors (ob.size());
            for (Json::Value::ArrayIndex i = 0; i !=
allow[(*iter)].size(); i++) {
                result_iter++;
                boost::dynamic_bitset<unsigned char> tst =
(*result_iter);
                actors |= tst;
            }
            violations |= ~(~(ob) | actors);
            result_iter++;
```

```
        }
        std::string result_rpc = "";
        int count = violations.count();
        result_rpc += std::to_string(count) + " entries violates your
privacy";
        if(count > 0) {
            std::string violations_str;
            boost::to_string(violations, violations_str);
            result_rpc += "\nIndexes (1 equals privacy breach): " +
              violations_str;

        }
        return result_rpc;
    }
```

4.7 VERIFICADOR

O Verificador recebe o arquivo com os *indexes* e *keywords* cifrados com algoritmo de criptografia consultável e uma *string* com a *query* a ser realizada e retorna um arquivo com o resultado final. Para isso, o verificador precisa instanciar um objeto do *Server* da biblioteca *OpenSSE* iniciar os *indexes* e *keywords* cifrados no objeto com o arquivo criado pelo Sistema Médico e então realizar a *query* enviada pelo cliente. Isso irá gerar um arquivo de resultado, como explicado na seção 4.2, que deve ser enviado para o cliente para gerar os resultados.

```
    [
      {
        "name": "SearchQuery",
        "params": {
          "query": "",
          "edb_path": "",
          "output": ""
        },
        "returns" : false
```

```
    }
]
```

A execução da *SearchQuery* precisa da localização do arquivo com os *indexes* e *keywords* cifrados, a *string* com a *query* a ser realizada e a localização para o arquivo de saída. O *SearchQuery* inicializa uma instância do *Server* da *OpenSSE* e então faz a busca com a *query recebida* para o arquivo na localização especificada nos parâmetros. A função está descrita abaixo.

```
bool VerifierServer::SearchQuery(const std::string& query, const
std::string& edb_path, const std::string& output)
{
Server *s = new Server();
s->SetupEDB(edb_path);
s->conjunctiveSearch(query, output);
return true;
}
```

4.8 APLICAÇÃO WEB

Foram desenvolvidas três páginas *web* que chamam as funções executadas por esses três serviços. A página do Verificador precisa de três entradas de texto são elas: a *query*, a localização do *Encrypted Database* e o caminho para o arquivo de saída. Além disso, precisa de um botão para executar a verificação. Uma foto desta tela está representada na Figura 13.

Figura 13 - Página Web do Verificador

A página do sistema médico precisa de um campo para retornar as preferências do usuário e um campo para escolher o caminho do arquivo de saída do *Encrypted Database*. Além disso, precisa de dois botões, um para chamar a função que retorna as preferências e outro para executar a função que cria o *Encrypted Database*. A página do sistema médico está representada na Figura 14.

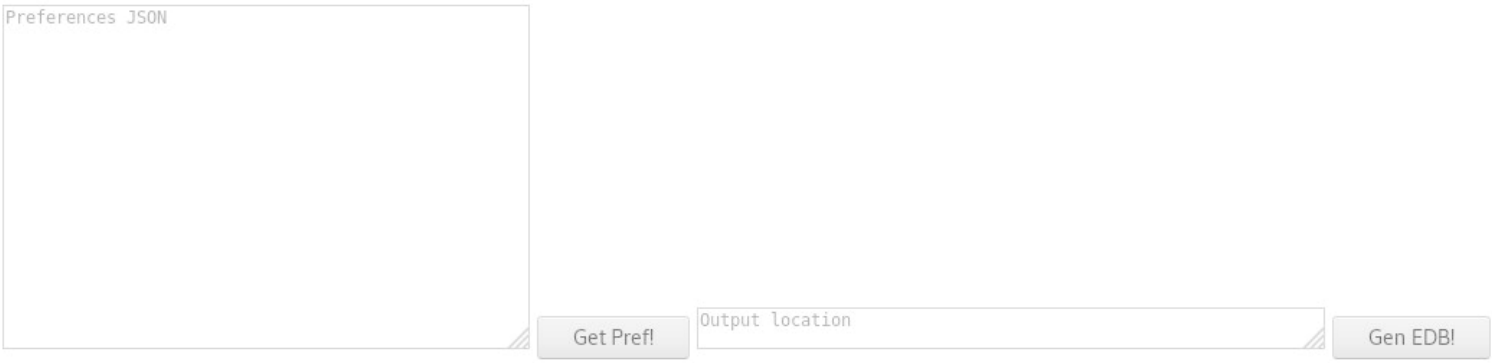

Figura 14 - Página Web do Sistema Médico

A página do Paciente deve ter duas entradas de texto, são elas: as preferências do usuário para geração da *query,* a localização do resultado do verificador. Além disso, ela precisa mostrar a *query* gerada a partir das preferências. A página possui um botão para geração da query e um para a verificação dos resultados. A página está representada pela Figura 15.

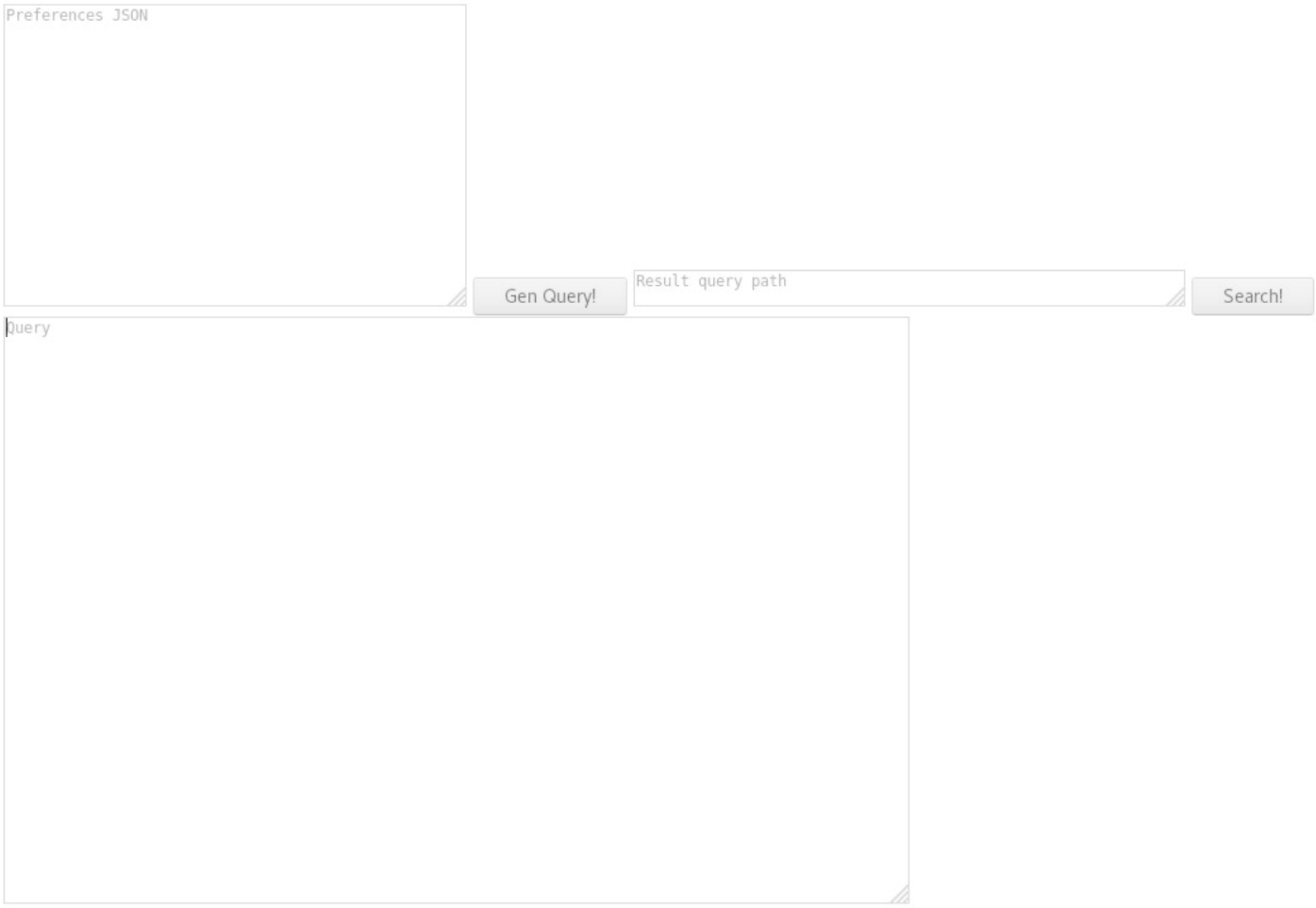

Figura 15 - Página do Paciente

A página do paciente precisa também alertar o usuário sobre o resultado da verificação. Para isso, ela abre uma janela dizendo se houve ou não violação da privacidade do usuário, e se houver, mostra uma lista dos índices com 1's e 0's onde 1 é um índice que houve violação das preferências. Essas janelas estão representadas nas Figuras 16 e 17.

4.9 TESTES E EXPERIMENTOS

Foram realizados testes com um número pequeno de logs para verificar a consistência dos resultados. Não foi levada em consideração a eficiência do algoritmo de criptografia consultável, por isso, foram utilizados somente 5 logs para cada teste. Ao executar os passos necessários pelas janelas do Sistema Médico, Paciente e Verificador a página do Verificador deve informar se houve ou não violação das preferências de acesso, e caso houver, mostrar uma lista dos índices de quais logs violaram as preferências.

No primeiro teste foram utilizados os seguintes logs de acesso:

ACTION:GET DATE:2018-02-14 ACTOR:1 OB:URINE-SAMPLE
ACTION:GET DATE:2018-02-14 ACTOR:2 OB:X-RAY
ACTION:GET DATE:2018-02-14 ACTOR:3 OB:X-RAY
ACTION:GET DATE:2018-02-14 ACTOR:4 OB:X-RAY

De acordo com as preferências definidas na seção 4.4, há violação no primeiro registro de acesso, onde o *ACTOR:1* acessa as informações de *URINE-SAMPLE,* a qual só permite acessos dos *ACTORS* 2 e 3.

A página do Paciente gera a *query* escrita abaixo.

9E8F7DBBF548DB6844B616E5736FC8175FC34993:465E69F2FECED834:53E390BCB6E3
1EEC;93FA068FE620B9DC98BEF22132CD2F3EF5AD0536:7102DA650533E78C;DF80438
9ABBE11EADF02C7CC7EF379EEC0BD9470:7102DA650533E78C;538CE4BB82E4295BC3
7ACBFBC1FEB2DED814F8C6:7102DA650533E78C;1F5191BD1E25D8BADD28DC1ADAA
1DCBB234EA936:7102DA650533E78C;33C715E61BC41C126955AB1A03F1C4C8857670F
F:7102DA650533E78C;1F5191BD1E25D8BADD28DC1ADAA1DCBB234EA936:7102DA65
0533E78C;BB3DC21545B3681307859F8F7875D6676A844E65:7102DA650533E78C;1B8A8
7F08DF47F42CBCCAB0CDDF4675E8E4ABFE4:7102DA650533E78C;4111872C5371ED26
5EC125DF9E4074885E113501:7102DA650533E78C;906D2E2AE2AAB34369D06ADAA22C

4EFE5466C9BB:7102DA650533E78C;E161128ACC5EA6AF0F1DF2ED8298BC120675E37F:7102DA650533E78C;28239C7972038552E87E6755FE2EB0E64D416C37:7102DA650533E78C;906D2E2AE2AAB34369D06ADAA22C4EFE5466C9BB:7102DA650533E78C;E161128ACC5EA6AF0F1DF2ED8298BC120675E37F:7102DA650533E78C;7AF2BD12F3DFCE8D367F782B3B36D2CF2D16AEA2:7102DA650533E78C;028285D1E0F48C04FA8D68022E16BFA6A51F55BB:7102DA650533E78C;2C429CEA247F69BCCA9CE507CBC379DF283C0695:7102DA650533E78C;DDAE2DDD4B30E7D8E597569B95B356063C13CF9A:7102DA650533E78C;906D2E2AE2AAB34369D06ADAA22C4EFE5466C9BB:7102DA650533E78C;D7F2C6B217B1C532421B2D26337F4CCFFDDC0558:7102DA650533E78C;1F5191BD1E25D8BADD28DC1ADAA1DCBB234EA936:7102DA650533E78C;D27C5B6ED5E423F4A19F8E38FD158D019B8D879F:7102DA650533E78C;906D2E2AE2AAB34369D06ADAA22C4EFE5466C9BB:7102DA650533E78C;028285D1E0F48C04FA8D68022E16BFA6A51F55BB:7102DA650533E78C;2C429CEA247F69BCCA9CE507CBC379DF283C0695:7102DA650533E78C;E161128ACC5EA6AF0F1DF2ED8298BC120675E37F:7102DA650533E78C;

Essa *query* deve ser enviada junto com o *Encrypted Database,* para o verificador que escreve um arquivo com os resultados. Estes resultados devem ser decifrados pela página do paciente a qual retornará a informação sobre a violação das preferências de acesso. Neste caso houve violação no log de índice 1 na lista dos logs. Essa resposta está representada na Figura 16.

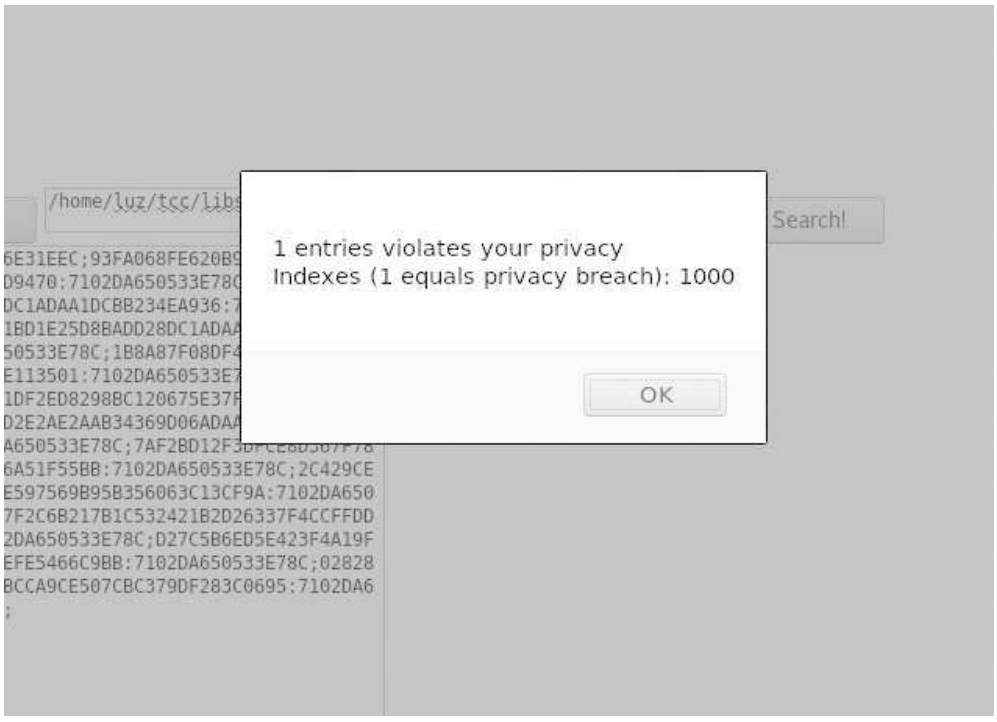

Figura 16 - Resultado violação das preferências

Outro teste realizado foi com logs onde não há violação das preferências do usuário, simplesmente trocando o *ACTOR* no log que violava para um permitido.

ACTION:GET DATE:2018-02-14 ACTOR:2 OB:URINE-SAMPLE
ACTION:GET DATE:2018-02-14 ACTOR:2 OB:X-RAY
ACTION:GET DATE:2018-02-14 ACTOR:3 OB:X-RAY
ACTION:GET DATE:2018-02-14 ACTOR:4 OB:X-RAY

Já que a *query* depende somente das preferências do usuário, ela se mantém igual ao primeiro teste. Ao final, a verificação do resultado pela página do paciente retornará a mensagem de que 0 logs violam as preferências, mostrado na Figura 17.

Figura 17 - Resultado não violação das preferências

“I may not have gone where I intended to go, but I think I have ended up where I needed to be.”
― Douglas Adams, The Long Dark Tea-Time of the Soul

5 CONCLUSÃO

O uso de *searchable encryption* fora do contexto padrão da interação entre o cliente e a nuvem é muito interessante e promissor. Este trabalho proporcionou o desenvolvimento de uma biblioteca de criptografia consultável que torna possível explorar o uso de criptografia consultável em outras áreas.

Com a biblioteca OpenSSE, a implementação do sistema médico, cliente e principalmente a do verificador foram objetivas e claras. Com isso, seria simples outras entidades utilizarem o mesmo protocolo de criptografia consultável e comunicação definidos neste trabalho para implementar outros verificadores, aumentando a confiabilidade na verificação feitas pelos usuários de sistemas médicos.

Com o uso de um esquema de *searchable encryption* que não precise de uma verificação dos resultados pelo cliente teria um impacto maior pois poderiam ser feitas arguições automáticas de dados disponíveis pelo sistema médico, como os dados cifrados de todos os clientes junto com a query que representa suas preferências.

A entidade que decida utilizar este esquema para prover uma maior confiabilidade de seu sistema tem que garantir a integridade da geração de logs de acesso assim como a sua integridade na geração dos logs cifrados. A confiabilidade do cliente nos verificadores pode ser melhorada com o número de implementações de verificadores e ele possa utilizar mais de um. O *software* que o cliente utiliza para criar a *query* e verificar o resultado também pode ser implementado em diferentes lugares além do que no próprio sistema médico. A confiabilidade no software que o cliente tem também pode ser melhorada por diferentes implementações disponíveis.

Este trabalho também poderia ser integrado a um *framework* de sistema médico para atingir diversas implementações consequentemente um número maior de usuários. Um *frameworks* de sistema médico disponível é o CareKit da Apple. Em 2017 a Apple lançou a possibilidade do compartilhamento de dados médicos na sua framework, porém depende da empresa implementando o sistema médico para definir políticas de privacidade e prover o usuário a capacidade de verificar essas políticas (DEKOSHKA, 2017). Com a integração deste trabalho com o *framework* desenvolvido pela Apple, seria possível

transformar a implementação de qualquer sistema médico utilizando o CareKit seguro, fortalecendo privacidade dos usuários. Além disso, o CareKit é uma framework open-source e busca a colaboração de desenvolvedores como houve colaborações no desenvolvimento do compartilhamento de dados médicos (DEKOSHKA, 2017).

7.1 TRABALHOS FUTUROS

Com o desenvolvimento deste trabalho, alguns pontos interessantes foram levados para serem desenvolvidos posteriormente. O primeiro deles, seria a implementação de outros algoritmos de criptografia consultável para a biblioteca OpenSSE. Com o surgimento de novos esquemas de criptografia consultável. O segundo, seria o estudo da aplicação de *searchable encryption* no contexto mais tradicional de banco de dados, a utilização da biblioteca OpenSSE seria muito interessante em uma implementação de um banco de dados SQL com suporte a *searchable encryption* para o cenário clássico dessa área entre cliente e nuvem. O terceiro, seria o estudo de novos esquemas de *searchable encryption* que não dependam de *response-hiding* para consultas booleanas expressivas. Dessa maneira seria possível eliminar a última interação com o cliente para validar o resultado neste trabalho.

REFERÊNCIAS

BBC News. **Edward Snowden: Leaks that exposed US spy programme**: 2016. Disponível em: <http://www.bbc.com/news/world-us-canada-23123964 >. Acesso em: 20 out. 2017.

CNBC. **Apple is quietly working on turning your iPhone into the one-stop shop for all your medical info**: 2017. Disponível em: <https://www.cnbc.com/2017/06/14/apple-iphone-medical-record-integration-plans.html>. Acesso em 10 jan. 2018

SWEENEY, Latanya. k-anonymity: A model for protecting privacy. **International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems**, v. 10, n. 05, p. 557-570, 2002.

ACAR, Abbas et al. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. **arXiv preprint arXiv:1704.03578**, 2017.

RIVEST, Ronald L.; SHAMIR, Adi; ADLEMAN, Leonard. A method for obtaining digital signatures and public-key cryptosystems. **Communications of the ACM**, v. 21, n. 2, p. 120-126, 1978.

RIVEST, Ronald L.; ADLEMAN, Len; DERTOUZOS, Michael L. On data banks and privacy homomorphisms. **Foundations of secure computation**, v. 4, n. 11, p. 169-180, 1978.

IDALINO, Thaís Bardini; SPAGNUELO, Dayana; MARTINA, Jean Everson. Private Verification of Access on Medical Data: An Initial Study. In: **Data Privacy Management, Cryptocurrencies and Blockchain Technology**. Springer, Cham, 2017. p. 86-103.

KATZ, Jonathan et al. **Handbook of applied cryptography**. CRC press, 1996.

ABDELRAHEEM, Mohamed Ahmed et al. Executing boolean queries on an encrypted bitmap index. In: **Proceedings of the 2016 ACM on Cloud Computing Security Workshop**. ACM, 2016. p. 11-22.

SUTTON, Andrew; SAMAVI, Reza. Blockchain enabled privacy audit logs. In: **International Semantic Web Conference**. Springer, Cham, 2017. p. 645-660.

DIFFIE, Whitfield; HELLMAN, Martin. New directions in cryptography. **IEEE transactions on Information Theory**, v. 22, n. 6, p. 644-654, 1976.

CNBC. **Apple is quietly working on turning your iPhone into the one-stop shop for all your medical info**: 2017. Disponível em: <https://www.cnbc.com/2017/06/14/apple-iphone-medical-record-integration-plans.html>. Acesso em 10 jan. 2018

DEKOSHKA, K. **Connecting CareKit to the Cloud**: 2017. WWDC. Disponível em: <https://developer.apple.com/videos/play/wwdc2017/239/>. Acesso em 10 jan. 2018.

EARL, J. **Building Apps with ResearchKit:** 2015. WWDC. Disponível em: <https://developer.apple.com/videos/play/wwdc2015/213/> . Acesso em 10 jan. 2018

UMER. **Getting Started with Carekit**: 2016. WWDC. Disponível em: <https://developer.apple.com/videos/play/wwdc2016/237/>. Acesso em 10 jan. 2018

MATT. **Getting the Most Out of HealthKit:** 2016. WWDC. Disponível em: <https://developer.apple.com/videos/play/wwdc2016/209/>. Acesso em 10 jan. 2018

BÖSCH, Christoph et al. A survey of provably secure searchable encryption. **ACM Computing Surveys (CSUR)**, v. 47, n. 2, p. 18, 2015.

CASH, David et al. Highly-scalable searchable symmetric encryption with support for boolean queries. In: **Advances in cryptology–CRYPTO 2013**. Springer, Berlin, Heidelberg, 2013. p. 353-373.

KAMARA, Seny; MOATAZ, Tarik. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: **Annual**

International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Cham, 2017. p. 94-124.

BENDER, Duane; SARTIPI, Kamran. HL7 FHIR: An Agile and RESTful approach to healthcare information exchange. In: **Computer-Based Medical Systems (CBMS), 2013 IEEE 26th International Symposium on**. IEEE, 2013. p. 326-331.

LOHR, Steve. **Google Is Closing Its Health Records Service**: 2011. Disponível em: <https://www.nytimes.com/2011/06/25/technology/25health.html> . Acesso em 20 mai. 2018

The Economist. **The vault is open**: 2007. Disponível em: <https://www.economist.com/node/9916512>. Acesso em: 20 mai. 2018.

Markle Foundation. **Connecting for Health: A Public-Private Collaborative**: 2003. Disponível em: <https://web.archive.org/web/20070104212409/http://www.connectingforhealth.org/resources/final_phwg_report1.pdf>. Acesso em 30 jun. 2018.

KIERKEGAARD, Patrick. Electronic health record: Wiring Europe's healthcare. **Computer law & security review**, v. 27, n. 5, p. 503-515, 2011.

TANG, Paul C. et al. Personal health records: definitions, benefits, and strategies for overcoming barriers to adoption. **Journal of the American Medical Informatics Association**, v. 13, n. 2, p. 121-126, 2006.

APÊNDICE A – Logs de acesso

Log com acesso fora das preferências:
ACTION:GET DATE:2018-02-14 ACTOR:1 OB:URINE-SAMPLE
ACTION:GET DATE:2018-02-14 ACTOR:2 OB:X-RAY
ACTION:GET DATE:2018-02-14 ACTOR:3 OB:X-RAY
ACTION:GET DATE:2018-02-14 ACTOR:4 OB:X-RAY

Log sem acesso fora das preferências:
ACTION:GET DATE:2018-02-14 ACTOR:2 OB:URINE-SAMPLE
ACTION:GET DATE:2018-02-14 ACTOR:2 OB:X-RAY
ACTION:GET DATE:2018-02-14 ACTOR:3 OB:X-RAY
ACTION:GET DATE:2018-02-14 ACTOR:4 OB:X-RAY

APÊNDICE B – Código Fonte OpenSSE

Client.hpp

```
#ifndef CLIENT_HH
#define CLIENT_HH

#include <boost/dynamic_bitset.hpp>
#include "Config.hpp"
#include <boost/filesystem.hpp>
#include <boost/asio.hpp>
#include <boost/array.hpp>
#include <unordered_map>
#include <deque>
#include <boost/random.hpp>

using boost::asio::ip::tcp;
typedef boost::mt19937 base_generator_type;

class Client {

private:

   std::vector<std::string>* words;
   std::unordered_map<std::string, int>* wordIndexes;

   unsigned int nrOfDocs;
   unsigned int id;

   //Indexing a document set
         void parseFile(std::string p, unsigned int limit, std::vector<boost::dynamic_bitset<unsigned char>>* bitsets);
      void buildVectors(std::string path, unsigned int limit, std::vector<boost::dynamic_bitset<unsigned char>>* bitsets);

   //Security functions
```

```
                                                                    void
permuteVectors(std::vector<boost::dynamic_bitset<unsigned    char>>*
bitsets);
        std::string encrypt(std::string plain, std::string key);
        std::string generateHash(std::string tag, std::string key);
        std::string generateRandomKey(int len);
                        boost::dynamic_bitset<unsigned       char>*
generateBitmapKey(std::string baseKey, int size);
                        boost::dynamic_bitset<unsigned       char>*
generateBitmapKeyBlock(std::string baseKey, int id);
                void    encryptDatabase(std::unordered_map<std::string,
std::string>* stags, std::unordered_map<std::string, std::string>* xtags,
std::vector<boost::dynamic_bitset<unsigned char>>* bitsets);

        std::vector<int>* makeIntegerList(const std::string &ids);
        std::vector<std::string> fetchKeys(std::string word);

                  void    WriteDatabase(std::unordered_map<std::string,
std::string>* stags, std::unordered_map<std::string, std::string>* xtags,
std::vector<boost::dynamic_bitset<unsigned char>>* bitsets, std::string
output);


    public:
        Client();
        ~Client();
          void setupDB(std::string path, unsigned int limit, std::string
output);
        std::string singleSearch(std::string word);
            std::string conjunctiveSearch(std::string sterm, std::string
xterms);
                  std::vector<boost::dynamic_bitset<unsigned    char>>
fetchResult(std::string result_file);
        void clear();
    };

    #endif
```

Client.cpp

```
#include "sse/Client.hpp"
#include <fstream>
#include <sstream>
#include <string>
#include <iostream>
#include <cmath>

#include <botan/botan.h>
#include <botan/hmac_rng.h>
#include <botan/gost_3411.h>
#include <botan/hmac.h>
#include <botan/bcrypt.h>
#include <botan/pipe.h>
#include <botan/key_filt.h>
#include <botan/hex_filt.h>
#include <botan/basefilt.h>
#include <botan/filters.h>

#include <boost/random/linear_congruential.hpp>
#include <boost/random/uniform_int.hpp>
#include <boost/random/uniform_real.hpp>
#include <boost/random/variate_generator.hpp>
#include <boost/generator_iterator.hpp>


Client::Client()
{
   this->id = 0;
   this->nrOfDocs = 0;
}

Client::~Client()
{
   if(this->words != nullptr)
      delete this->words;
```

```
            if(this->wordIndexes != nullptr)
                delete this->wordIndexes;
        }

        void Client::setupDB(std::string path, unsigned int limit, std::string output)
        {
            std::cout << "Setting up encrypted database..." << std::endl;

             std::vector<boost::dynamic_bitset<unsigned char>>* bitsets = new std::vector<boost::dynamic_bitset<unsigned char>>();
            this->words = new std::vector<std::string>();
             this->wordIndexes = new std::unordered_map<std::string, int>();

            std::cout << "Building bitmaps...";
            buildVectors(path, limit, bitsets);
            std::cout << "OK" << std::endl;

              delete this->wordIndexes;//No longer need it, used for fast access of words
            this->wordIndexes = nullptr;
             this->nrOfDocs = this->id;//All files have been traversed and id is the last index
                std::unordered_map<std::string, std::string>* stags = new std::unordered_map<std::string, std::string>(this->words->size());
                std::unordered_map<std::string, std::string>* xtags = new std::unordered_map<std::string, std::string>(this->words->size());

            permuteVectors(bitsets);

            std::cout << "Encrypting index, building tags...";
            encryptDatabase(stags, xtags, bitsets);
            std::cout << "OK" << std::endl;

            delete this->words;//No longer needed
            this->words = nullptr;
```

```
            std::cout << "Writing encrypted database to file...";
            WriteDatabase(stags, xtags, bitsets, output);
            std::cout << "OK" << std::endl;

            std::cout << "Done" << std::endl;
        }

        void Client::WriteDatabase(std::unordered_map<std::string, std::string>* stags, std::unordered_map<std::string, std::string>* xtags, std::vector<boost::dynamic_bitset<unsigned char>>* bitsets, std::string output)
        {
            boost::filesystem::ofstream myfile;
            myfile.open(output);
            std::string req = std::to_string(functions::START_SETUP);
            std::stack<boost::dynamic_bitset<unsigned char>> queue;
            int size = bitsets->size();
            myfile << req << FUNC_DELIM;
            for(int a = size-1; a >= 0; a--) {
                queue.push((*bitsets)[a]);
                bitsets->erase(bitsets->begin() + a);
            }
            delete bitsets;
            bitsets = nullptr;

            //Send all bitmaps
            for(int a = 0; a < size; a++) {
                req = "";
                boost::to_string(queue.top(), req);
                queue.pop();
                req = std::to_string(functions::ADD_DB) + req;
                myfile << req << FUNC_DELIM;
            }
            req = std::to_string(functions::END_BITMAPS);
            myfile << req << FUNC_DELIM;

            //Send all stags
            for(auto iter = stags->begin(); iter != stags->end(); ++iter) {
```

```
            req = std::to_string(functions::ADD_DB);
            req += iter->first;
            req += ARG_DELIM;
            req += iter->second;
            myfile << req << FUNC_DELIM;
        }
        req = std::to_string(functions::END_STAGS);
        myfile << req << FUNC_DELIM;
        delete stags;
        stags = nullptr;

        //Send all xtags
        for(auto iter = xtags->begin(); iter != xtags->end(); ++iter) {
            req = std::to_string(functions::ADD_DB);
            req += iter->first;
            req += ARG_DELIM;
            req += iter->second;
            myfile << req << FUNC_DELIM;
        }
        req = std::to_string(functions::END_XTAGS);
        myfile << req << FUNC_DELIM;
        myfile.close();
        delete xtags;
        xtags = nullptr;
    }
    void Client::encryptDatabase(std::unordered_map<std::string,
std::string>* stags, std::unordered_map<std::string, std::string>* xtags,
std::vector<boost::dynamic_bitset<unsigned char>>* bitsets)
    {
        std::string s_key, is_key, x_key, ix_key, b_key, index, word;
        for(unsigned int a = 0; a < this->words->size(); a++)
        {
            word = (*this->words)[a];
            // Call local function to read in the keys for the current word
            std::vector<std::string> keys = fetchKeys(word);
            //Generate all keys for this word if they do not exist
            if(keys.size() == 0)
            {
```

```
                //Open a file with the name of the keyword
                std::ofstream out (KEYS+word);
                s_key = generateRandomKey(8);
                is_key = generateRandomKey(8);
                x_key = generateRandomKey(8);
                ix_key = generateRandomKey(8);
                b_key = generateRandomKey(8);
                 out << s_key << " " << is_key << " " << x_key << " " <<
ix_key << " " << b_key << std::endl;
                out.close();
            }
            //Compute stag and xtag
            index = std::to_string(a);

             stags->insert({generateHash(word, keys[0]), encrypt(index,
keys[1])});
             xtags->insert({generateHash(word, keys[2]), encrypt(index,
keys[3])});
            //Make sure bitmap is of right size

            (*bitsets)[a].resize(this->nrOfDocs);

            //Generate bitmap key and encrypt
                        boost::dynamic_bitset<unsigned char>* key =
generateBitmapKey(keys[4], this->nrOfDocs);
            (*bitsets)[a] ^= *key;
            delete key;
            key = nullptr;
          }
        }
        void Client::buildVectors(std::string path, unsigned int limit,
std::vector<boost::dynamic_bitset<unsigned char>>* bitsets)
        {
           boost::filesystem::path logs_path(path);
            if(!boost::filesystem::exists(logs_path))//Check if file exists, if
not then exit
            {
```

```
                    std::cout << "File not found: " +
boost::filesystem::basename(logs_path) << std::endl;
                std::exit(401);
            }
            //Copy all entries from directory to the vector
            std::vector<boost::filesystem::path> logs_vector;
                 std::copy(boost::filesystem::directory_iterator(logs_path),
boost::filesystem::directory_iterator(), back_inserter(logs_vector));
            std::sort(logs_vector.begin(), logs_vector.end());
            //The indexes are being reversed somewhere, so reverse here
            std::reverse(logs_vector.begin(), logs_vector.end());

              for(auto iter = logs_vector.begin(); iter != logs_vector.end();
++iter) {
                if(this->id == limit)
                   return;
                if(boost::filesystem::is_regular_file(*iter))
                {
                                                    std::string absPath =
boost::filesystem::absolute((*iter)).string();
                   std::ifstream in (absPath, std::ios::in);
                   std::string line, temp;
                   if(in.is_open())
                   {
                      std::vector<std::string> lines_in_order;
                      while(std::getline(in, line))
                         lines_in_order.push_back(line);
                      //The indexes are being reversed somewhere, so reverse
here
                                 for (auto it = lines_in_order.rbegin(); it !=
lines_in_order.rend(); ++it)
                      {
                         parseFile((*it), limit, bitsets);
                      }
                   }
                   in.close();
                }
            }
```

```
        }

        void Client::parseFile(std::string p, unsigned int limit,
std::vector<boost::dynamic_bitset<unsigned char>>* bitsets)
        {
           typedef std::vector<std::string> vec;
           vec tempVec;
           vec::const_iterator it;
           unsigned int index;
           //Copy all words unsigned into a vector and iterate over
           std::stringstream ss(p);
                              std::copy(std::istream_iterator<std::string>(ss),
std::istream_iterator<std::string>(), std::back_inserter(tempVec));

           for(auto iter = tempVec.begin(); iter != tempVec.end(); ++iter)
           {
              auto wordIndexIter = this->wordIndexes->find(*iter);
              if(wordIndexIter == wordIndexes->end())
              {
                 //Word does not exist
                 this->words->push_back(*iter); //Add new word
                 boost::dynamic_bitset<unsigned char> bs (INIT_SIZE);
                 if(this->id >= INIT_SIZE)
                 {
                     bs.resize(this->id+1); //Sets the new size to make room
for new id plus the initial size
                 }
                 bs.set(this->id); //Set bit of current document ID
                 bitsets->push_back(bs); //Add new bitset
                 wordIndexes->insert({*iter, this->words->size()-1});
              }
              else
              {
                 //Word already exists
                 index = wordIndexIter->second;
                 if(this->id >= (*bitsets)[index].size())
                 {
                    (*bitsets)[index].resize(this->id + 1); //Resize bitset
```

```
                }
                (*bitsets)[index].set(this->id);
            }

        }
        this->id++;

    }

    void
Client::permuteVectors(std::vector<boost::dynamic_bitset<unsigned
char>>* bitsets) {
        //The same seed does not generate the same generator
        base_generator_type generator(BIT_SEED);
        boost::uniform_int<> uni_dist(0,this->words->size()-1);
                    boost::variate_generator<base_generator_type&,
boost::uniform_int<>> uni(generator, uni_dist);
        //Knuths shuffle
        for(unsigned int i = this->words->size()-1 ; i > 0 ; i--) {
            unsigned int index = uni() % (i);
            std::swap((*this->words)[i], (*this->words)[index]);
            std::swap((*bitsets)[i], (*bitsets)[index]);
        }
    }

    boost::dynamic_bitset<unsigned                              char>*
Client::generateBitmapKey(std::string baseKey, int size)
    {
        int iterations = std::ceil(size/SEC_PARAM);
        std::string temp = "";
        for(int a = 0; a < iterations; a++)
        {
            temp += generateHash(baseKey+std::to_string(a), baseKey);
        }

            boost::dynamic_bitset<unsigned char>* key = new
boost::dynamic_bitset<unsigned char>(0);
        key->append(temp.begin(), temp.end());
```

```
        key->resize(size);
        return key;
    }

    boost::dynamic_bitset<unsigned char>*
Client::generateBitmapKeyBlock(std::string baseKey, int id)
    {
            boost::dynamic_bitset<unsigned char>* key = new
boost::dynamic_bitset<unsigned char>(0);
        int a = id/SEC_PARAM;   //Gets the correct block value
          std::string temp = generateHash(baseKey+std::to_string(a),
baseKey);
        key->append(temp.begin(), temp.end());

        return key;
    }

    std::string Client::encrypt(std::string plain, std::string key)
    {
        Botan::SymmetricKey symkey (reinterpret_cast<Botan::byte
const *>(key.data()), key.size());
         Botan::InitializationVector iv (reinterpret_cast<Botan::byte
const *>(IV.data()), IV.size());
          Botan::Pipe pipe(Botan::get_cipher(ENC_ALGORITHM,
symkey, iv, Botan::ENCRYPTION), new Botan::Hex_Encoder);
        pipe.process_msg(plain);
        std::string e = pipe.read_all_as_string(0);
        return e;
    }

    std::string Client::generateRandomKey(int len)
    {
            Botan::HMAC_RNG mac(new Botan::HMAC(new
Botan::GOST_34_11), new Botan::HMAC(new Botan::GOST_34_11));
        Botan::SymmetricKey key (mac, len);
        return key.as_string();
    }
```

```
std::string Client::generateHash(std::string tag, std::string key)
{
                    Botan::SymmetricKey symkey = Botan::SymmetricKey(reinterpret_cast<Botan::byte const *>(key.data()), key.size());

                    Botan::Pipe pipe(new Botan::Chain(new Botan::MAC_Filter(HASH_ALGORITHM, symkey), new Botan::Hex_Encoder));

   pipe.process_msg(tag);
   return pipe.read_all_as_string(0);
}

std::vector<std::string> Client::fetchKeys(std::string word)
{
   std::ifstream in (KEYS+MASTER_KEY);
   std::string line;
   std::vector<std::string> keys;
   if(!in)
      return keys;

   std::getline(in, line);
   std::stringstream ss(line);
                    std::copy(std::istream_iterator<std::string>(ss), std::istream_iterator<std::string>(), std::back_inserter(keys));

   in.close();
   return keys;
}
std::string Client::singleSearch(std::string word)
{
   std::vector<std::string> keys = fetchKeys(word);
   if(keys.empty())
      return "";
   std::string s_key = keys[0];//Get the stag key
   std::string stag = generateHash(word, s_key); //Generate stag
   std::string functionID = std::to_string(functions::SKS);
```

```
            std::string i_key = keys[1];//Get the index key
            std::string b_key = keys[4];//Get the basekey for bitmap
            std::string result;

             //First string is for identifying function SKS, followed by stag,
i_key, b_key and bool telling Server whether to cache results
              std::string msg = functionID + stag + KEY_DELIM + i_key +
KEY_DELIM + b_key + ARG_DELIM;

            result = result.substr(0, result.find_first_of(END_OF_MSG));
            size_t n = std::count(result.begin(), result.end(), '1');
            std::cout << "Result " << n << std::endl;
            return result;
        }

        std::string Client::conjunctiveSearch(std::string sterm, std::string
xterms)
        {
            std::string result;
            std::string query, stag, functionID = "";
            int nrOfXterms;
            std::vector<std::string> xVec, keys;
                        std::vector<std::string>*     bitmapKeys     =     new
std::vector<std::string> ();

            //Add stag and its keys to query
            keys = fetchKeys(sterm);
            if(keys.empty())
            {
               std::cout << "did not find keys" << std::endl;
               delete bitmapKeys;
               return "";
            }

            stag = generateHash(sterm, keys[0]); //Generate stag
            functionID = std::to_string(functions::CKS);

            //The string will take the form: IDSTAG:I_KEY:B_KEY;
```

```
            query = stag + KEY_DELIM + keys[1] + KEY_DELIM +
keys[4] + ARG_DELIM;

            //Add xterms to query
            std::stringstream ss(xterms);
                              std::copy(std::istream_iterator<std::string>(ss),
std::istream_iterator<std::string>(), std::back_inserter(xVec));
            nrOfXterms = xVec.size();

            for(int a = 0; a < nrOfXterms; a++)
            {
               keys = fetchKeys(xVec[a]);
               if(keys.empty())
               {
                  std::cout << "did not find keys" << std::endl;
                  delete bitmapKeys;
                  return "";
               }
                bitmapKeys->push_back(keys[4]);//Save bitmap key so we
do not have read again
               query += generateHash(xVec[a], keys[2]) + KEY_DELIM +
keys[3] + ARG_DELIM; //Generate xtag, fetch index key and add to
query
            }

            return query;
         }

        std::vector<boost::dynamic_bitset<unsigned                char>>
Client::fetchResult(std::string result_file)
         {
            std::vector<std::string> keys;
                     std::vector<boost::dynamic_bitset<unsigned    char>>
result_x_tags;
            keys = fetchKeys(""); //Fetch key
            std::string bitmapkey = keys[4]; //Fetch bitmap key

            std::string bits, temp;
```

```
int count = 0;
bool bit, keyBit;

std::ifstream inFile;
inFile.open(result_file);//open the input file
std::stringstream strStream;
strStream << inFile.rdbuf();//read the file
std::string str = strStream.str();//str holds the content of the file
size_t pos = 0;
std::string token;
std::deque<std::string> queueIn;
while ((pos = str.find(FUNC_DELIM)) != std::string::npos) {
   token = str.substr(0, pos);
   queueIn.push_back(token);
   str.erase(0, pos + strlen(FUNC_DELIM));
}
//First, fetch the result of s-term
temp = queueIn.front();
temp = temp.substr(0, temp.find_first_of(END_OF_MSG));

size_t n = std::count(temp.begin(), temp.end(), '1');

std::vector<int>* intIds = makeIntegerList(temp);
if(intIds->empty())
   return result_x_tags;
queueIn.pop_front();
boost::dynamic_bitset<unsigned char> ids (temp);
boost::dynamic_bitset<unsigned char> sterm (temp);

while(!queueIn.empty()) { //X-tags
//Compare each xterm
   //Fetch the sent bits
   bits = queueIn.front();
   if(bits.length() == 0) {
      std::cout << "bits == 0 " << std::endl;
      return result_x_tags;
   }
   queueIn.pop_front();
```

```
            //Construct a bitmap of the bits
            boost::dynamic_bitset<unsigned char> bitmap (ids.size());
            if(bits != "201") //found xtag!
               for(int i : *intIds)
               {
                     boost::dynamic_bitset<unsigned char>* keyBlock =
generateBitmapKeyBlock(bitmapkey, i);
                  keyBit = keyBlock->test(i % SEC_PARAM);
                  delete keyBlock;
                  bit = (bits[count] == '1') != keyBit;
                  bitmap.set(i, bit);
                  count++;
               }
            count = 0;
            std::string strxterm;
            boost::to_string(bitmap, strxterm);
            result_x_tags.push_back(bitmap);
            size_t n2 = std::count(strxterm.begin(), strxterm.end(), '1');

         }

         delete intIds;
         intIds = nullptr;

         return result_x_tags;
      }

      std::vector<int>* Client::makeIntegerList(const std::string &ids)
      {
         if(ids.length() <= 1)//Empty string
            return new std::vector<int>();
         std::vector<int>* docIds = new std::vector<int> ();


         boost::dynamic_bitset<unsigned char> bitmap (ids);
         size_t bit = bitmap.find_first();
```

```
    for(;bit < bitmap.npos; bit = bitmap.find_next(bit))
    {
       //append to result
       docIds->push_back(bit);
    }
    return docIds;
}

void Client::clear()
{
    this->id = 0;
    this->nrOfDocs = 0;
}
```

Server.hpp

```
#ifndef SERVER_HH
#define SERVER_HH

#include <unordered_map>
#include <boost/dynamic_bitset.hpp>
#include <boost/array.hpp>
#include <boost/asio.hpp>
#include <deque>
#include <tuple>
#include <iostream>
#include "Config.hpp"
#include <boost/random.hpp>
#include <boost/filesystem.hpp>

class Server {

private:

    std::unordered_map<std::string, std::string> *xtags;
    std::unordered_map<std::string, std::string> *stags;
```

```
        std::vector<boost::dynamic_bitset<unsigned char>> *bitsets;

        std::vector<int> cachedSKS;
        bool cache;

          std::string decrypt(const std::string &cipher, const std::string
&key);
        boost::dynamic_bitset<unsigned char>* generateBitmapKey(
           const std::string &baseKey, int size);
             std::string generateHash(const std::string &word, const
std::string &key);

    public:

        Server();
        ~Server();

        void SetupEDB(std::string path);

        std::string singleKeywordSearch(std::string query);
              void conjunctiveSearch(std::string query, std::string
output_path);

        void clear();

    };

    #endif



    Server.cpp

    #include "sse/Server.hpp"
    #include <fstream>
    #include <iostream>
    #include <botan/pipe.h>
    #include <botan/key_filt.h>
```

```
#include <botan/hex_filt.h>
#include <botan/basefilt.h>
#include <botan/filters.h>
#include <botan/botan.h>

#include <boost/random/linear_congruential.hpp>
#include <boost/random/uniform_int.hpp>
#include <boost/random/uniform_real.hpp>
#include <boost/random/variate_generator.hpp>
#include <boost/generator_iterator.hpp>

Server::Server() {
   this->xtags = nullptr;
   this->stags = nullptr;
   this->bitsets = nullptr;
}

Server::~Server() {
   if(this->xtags != nullptr)
      delete this->xtags;
   if(this->stags != nullptr)
      delete this->stags;
   if(this->bitsets != nullptr)
      delete this->bitsets;
}

void Server::SetupEDB(std::string path) {
   std::deque<std::string> queueIn;
   std::ifstream inFile;
   inFile.open(path);//open the input file
   std::stringstream strStream;
   strStream << inFile.rdbuf();//read the file
   std::string str = strStream.str();//str holds the content of the file
   size_t pos = 0;
   std::string token;
   while ((pos = str.find(FUNC_DELIM)) != std::string::npos) {
      token = str.substr(0, pos);
      queueIn.push_back(token);
```

```
            str.erase(0, pos + strlen(FUNC_DELIM));
        }

        while(!queueIn.empty()) {
            std::string t = queueIn.front();
            queueIn.pop_front();
            int index = std::stoi(t.substr(0,3));
            bool recvBitmaps;
            bool recvStags;
            switch ( index ) {
                case functions::START_SETUP:
                        this->stags = new std::unordered_map<std::string,
std::string>();
                        this->xtags = new std::unordered_map<std::string,
std::string>();
                                                this->bitsets = new
std::vector<boost::dynamic_bitset<unsigned char>>();
                    recvBitmaps = true;
                    break;
                case functions::ADD_DB:
                    if(recvBitmaps) {
                                boost::dynamic_bitset<unsigned char> b
(t.substr(3));
                        this->bitsets->push_back(b);
                    } else if(recvStags) {
                        t = t.substr(3);
                        int delim = t.find_first_of(ARG_DELIM);
                        std::string stag = t.substr(0, delim);
                        std::string bIndex = t.substr(delim+1);
                        this->stags->insert({stag, bIndex});
                    } else {//ADD x-tags
                        t = t.substr(3);
                        int delim = t.find_first_of(ARG_DELIM);
                        std::string xtag = t.substr(0, delim);
                        std::string bIndex = t.substr(delim+1);
                        this->xtags->insert({xtag, bIndex});
                    }
                    break;
```

```
            case functions::END_BITMAPS:
                recvBitmaps = false;
                recvStags = true;
                break;

            case functions::END_STAGS:
                recvStags = false;
                break;

            default:
                break;
        }
    }
}

std::string Server::singleKeywordSearch(std::string query) {

    std::string stag, sendResult, final, temp, i_key, b_key = "";
    std::vector<int> result;
    int b_index;

    auto iter = query.begin(); //Start iterator

    while(*iter != KEY_DELIM) {//Stag
        stag += *iter;
        ++iter;
    }
    ++iter; //Step over key delimitor

    while(*iter != KEY_DELIM) { //First key
        i_key += *iter;
        ++iter;
    }
    ++iter; //Step over

    while(*iter != ARG_DELIM) { //Second key
        b_key += *iter;
        ++iter;
```

```
            }

            //Try to find S-TAG
            auto it = this->stags->find(stag);
            if(it == this->stags->end()) {
               std::cout << "S-TAG not found" << std::endl;
               return sendResult;
            }

            //Decrypt the bitmap index
            b_index = std::stoi(decrypt(it->second, i_key));

            //Fetch the bitmap using above index
                  boost::dynamic_bitset<unsigned char> bitmap =
(*this->bitsets)[b_index];
            //Generate the bitmap full key using the seed
                   boost::dynamic_bitset<unsigned char>* key =
generateBitmapKey(b_key, bitmap.size());

            //Decrypt
            bitmap ^= *key;
            delete key;

            //Iterate over to find active bits
            size_t bit = bitmap.find_first();
            for(;bit < bitmap.npos; bit = bitmap.find_next(bit))
               result.push_back(bit);

            //Check if results are to be saved for a conjunctive search
            if(this->cache)
               this->cachedSKS = result;

            //Turn bitmap into string
            std::string bRes;
            boost::to_string(bitmap, bRes);
            return bRes;
         }
```

```
void Server::conjunctiveSearch(std::string query, std::string output_path) {

        std::string tag, i_key, result, stag, bits, subMsg = "";
        int b_index;
        unsigned int a_delim = query.find_first_of(ARG_DELIM);
        int k_delim;

        boost::filesystem::ofstream output_file;
        output_file.open(output_path);
        //First, perform SKS
        this->cache = true;
            std::string sks = singleKeywordSearch(query.substr(0, a_delim+1));

        output_file << sks << FUNC_DELIM;
        this->cache = false;
        subMsg = query.substr(a_delim+1); //Excluding the delimitor
        while(subMsg.length() > 0)
        {
           a_delim = subMsg.find_first_of(ARG_DELIM);
           k_delim = subMsg.find_first_of(KEY_DELIM);
           tag = subMsg.substr(0, TAG_LEN);
           i_key = subMsg.substr(k_delim+1, KEY_LEN);

           // std::cout << "a_delim " << a_delim << std::endl;
           //Try to find the xtag
           auto iter = this->xtags->find(tag);
           if(iter == this->xtags->end())
           {
              result = std::to_string(functions::LAST_BIT);
              output_file << result << FUNC_DELIM;
              result = "";
              if(a_delim+1 >= subMsg.length())
                 subMsg = "";
              else
                 subMsg = subMsg.substr(a_delim+1);
              continue;
```

```
            }

            //Fetch the bitmap index of the xtag
            b_index = std::stoi(decrypt(iter->second, i_key));
            auto bitmap = (*this->bitsets)[b_index];

            //Fetch all decrypted bits at positions returned from SKS
            for(int id : this->cachedSKS)
               result += std::to_string(bitmap.test(id));

            output_file << result << FUNC_DELIM;

            //Prepare next iteration
            result = "";
            if(a_delim+1 >= subMsg.length()){
               // std::cout << "HEY" << std::endl;
               subMsg = "";
            }
            else
               subMsg = subMsg.substr(a_delim+1);
         }
         //Clear the saved result
         this->cachedSKS = std::vector<int> ();
      }

      std::string Server::decrypt(const std::string &cipher,
         const std::string &key) {

          Botan::SymmetricKey symkey (reinterpret_cast<Botan::byte
const *>(key.data()), key.size());
           Botan::InitializationVector iv (reinterpret_cast<Botan::byte
const *>(IV.data()), IV.size());
                    Botan::Pipe pipe(new Botan::Hex_Decoder,
get_cipher("AES-128/CBC/NoPadding", symkey, iv,
Botan::DECRYPTION));
         pipe.process_msg(cipher.c_str());
         std::string m = pipe.read_all_as_string(0);
```

```
        return m;
    }

    boost::dynamic_bitset<unsigned char>*
Server::generateBitmapKey(
        const std::string &baseKey, int size) {

        int iterations = std::ceil(size/SEC_PARAM);
        std::string temp = "";
        for(int a = 0; a < iterations; a++)
        {
            temp += generateHash(baseKey+std::to_string(a), baseKey);
        }

            boost::dynamic_bitset<unsigned char>* key = new
boost::dynamic_bitset<unsigned char>(0);
        key->append(temp.begin(), temp.end());
        key->resize(size);

        return key;
    }

    std::string Server::generateHash(const std::string &word,
        const std::string &key) {

                    Botan::SymmetricKey symkey =
Botan::SymmetricKey(reinterpret_cast<Botan::byte const
*>(key.data()), key.size());

                Botan::Pipe pipe(new Botan::Chain(new
Botan::MAC_Filter(HASH_ALGORITHM, symkey), new
Botan::Hex_Encoder));

        pipe.process_msg(word);
        return pipe.read_all_as_string(0);
    }

    void Server::clear() {
```

```
        if(this->xtags != nullptr)
            delete this->xtags;
        if(this->stags != nullptr)
            delete this->stags;
        if(this->bitsets != nullptr)
            delete this->bitsets;

        this->xtags = nullptr;
        this->stags = nullptr;
        this->bitsets = nullptr;

    }
```

Config.hpp

```
    #ifndef CONFIG_HH
    #define CONFIG_HH

    //Enum for functions in Server
    enum functions{CKS=100, SKS, START_SETUP,
ADD_DB=200, LAST_BIT, END_BITMAPS, END_STAGS,
END_XTAGS, CLEAR=300};

    //Delimitor for variables in message sent to Server
    const char ARG_DELIM = ';';
    const char KEY_DELIM = ':';
    const char END_OF_MSG = '.';
    static const char* const FUNC_DELIM = "\x1B";

    //Initial size of bitsets
    const int INIT_SIZE = 5;

    //Path to directory holding keys
    const std::string KEYS =
"/home/luz/tcc/libssebitmap/resources/keys/";

    //Global security parameters
```

```
const int BIT_SEED = 123481;
const std::string IV = "kj43hSDF23kjnkfF"; //16 byte IV
const std::string ENC_ALGORITHM = "AES-128/CBC";
const std::string HASH_ALGORITHM = "HMAC(SHA-1)";
const int SEC_PARAM = 1; //For calculating bitmapkey

//Primarily used when extracting keys and tags
const int KEY_LEN = 16;
const int TAG_LEN = 40;

//For testing purposes
const std::string MASTER_KEY = "MASTER";

#endif
```

APÊNDICE C – Código Fonte Verificador

VerifierSpec.json

```
[
   {
      "name": "SearchQuery",
      "params": {
         "query": "",
         "edb_path": "",
         "output": ""
      },
      "returns" : false
   }
]
```

AbstractStubVerifierServer.h

```
/**
 * This file is generated by jsonrpcstub, DO NOT CHANGE IT MANUALLY!
 */

#ifndef JSONRPC_CPP_STUB_ABSTRACTSTUBVERIFIERSERVER_H_
#define JSONRPC_CPP_STUB_ABSTRACTSTUBVERIFIERSERVER_H_

#include <jsonrpccpp/server.h>

class AbstractStubVerifierServer : public jsonrpc::AbstractServer<AbstractStubVerifierServer>
{
   public:

AbstractStubVerifierServer(jsonrpc::AbstractServerConnector &conn, jsonrpc::serverVersion_t type = jsonrpc::JSONRPC_SERVER_V2) : jsonrpc::AbstractServer<AbstractStubVerifierServer>(conn, type)
        {
```

```
this->bindAndAddMethod(jsonrpc::Procedure("SearchQuery",
jsonrpc::PARAMS_BY_NAME, jsonrpc::JSON_BOOLEAN,
"edb_path",jsonrpc::JSON_STRING,"output",jsonrpc::JSON_STRING,
"query",jsonrpc::JSON_STRING, NULL),
&AbstractStubVerifierServer::SearchQueryI);
        }

                inline virtual void SearchQueryI(const Json::Value
&request, Json::Value &response)
        {
                                                response =
this->SearchQuery(request["edb_path"].asString(),
request["output"].asString(), request["query"].asString());
        }
         virtual bool SearchQuery(const std::string& edb_path, const
std::string& output, const std::string& query) = 0;
    };

    #endif
//JSONRPC_CPP_STUB_ABSTRACTSTUBVERIFIERSERVER_H_
```

VerifierServer.cpp

```
#include "stub/AbstractStubVerifierServer.h"
#include <jsonrpccpp/server/connectors/httpserver.h>
#include <iostream>
#include <stdio.h>
#include "sse/Server.hpp"

using namespace jsonrpc;
using namespace std;

class VerifierServer : public AbstractStubVerifierServer
{
  public:
    VerifierServer(AbstractServerConnector &connector);
```

```
            virtual bool SearchQuery(const std::string& query, const
std::string& edb_path, const std::string& output);
    };

    VerifierServer::VerifierServer(AbstractServerConnector
&connector) :
        AbstractStubVerifierServer(connector)
    {
    }

    bool VerifierServer::SearchQuery(const std::string& query, const
std::string& edb_path, const std::string& output)
    {
        Server *s = new Server();
        s->SetupEDB(edb_path);
        s->conjunctiveSearch(query, output);
        return true;
    }

    int main()
    {
        HttpServer httpserver(8383);
        VerifierServer s(httpserver);
        s.StartListening();
        getchar();
        s.StopListening();
        return 0;
    }
```

StubVerifierClient.js

```
    /**
     * This file is generated by jsonrpcstub, DO NOT CHANGE IT
MANUALLY!
     */
    function Verifier(url) {
```

```
        this.url = url;
        var id = 1;

            function doJsonRpcRequest(method, params, methodCall,
callback_success, callback_error) {
            var request = {};
            if (methodCall)
               request.id = id++;
            request.jsonrpc = "2.0";
            request.method = method;
            if (params !== null) {
               request.params = params;
            }
            JSON.stringify(request);

            $.ajax({
               type: "POST",
               url: url,
               data: JSON.stringify(request),
               success: function (response) {
                  if (methodCall) {
                              if (response.hasOwnProperty("result") &&
response.hasOwnProperty("id")) {
                     callback_success(response.id, response.result);
                  } else if (response.hasOwnProperty("error")) {
                     if (callback_error != null)

callback_error(response.error.code,response.error.message);
                  } else {
                     if (callback_error != null)
                                 callback_error(-32001, "Invalid Server
response: " + response);
                  }
               }
            },
            error: function () {
               if (methodCall)
                  callback_error(-32002, "AJAX Error");
```

```
            },
            dataType: "json"
        });
        return id-1;
    }
        this.doRPC = function(method, params, methodCall,
callback_success, callback_error) {
        return doJsonRpcRequest(method, params, methodCall,
callback_success, callback_error);
    }
}

Verifier.prototype.SearchQuery = function(edb_path, output,
query, callbackSuccess, callbackError) {
    var params = {edb_path : edb_path, output : output, query :
query};
        return this.doRPC("SearchQuery", params, true,
callbackSuccess, callbackError);
};
```

verifier.html

```
<!DOCTYPE html>
<html>
<head>
<meta content="text/html;charset=utf-8"
http-equiv="Content-Type">
<title>Verifier</title>
<script src="http://code.jquery.com/jquery-2.1.1.min.js"
type="text/javascript"></script>
<script src="js/StubVerifierClient.js"
type="text/javascript"></script>
<script type="text/javascript">
$(document).ready(function () {
  function showResult(id, result) {
```

```
            alert(" Result: " + result);
        }
        function showError(code, message) {
            alert("Error: " + code + " -> " + message);
        }

        $('#button').click(function() {
            var verifier = new Verifier("http://localhost:8383");
                                verifier.SearchQuery($('#query').val(),
$('#edb').val(),$('#output').val() ,showResult, showError);
        });
    });
    function displayResult(result) {
        alert(JSON.stringify(result));
    }
    </script>
    </head>
    <body>
            <textarea id="query"placeholder="Input your query here"
cols="100" rows="30"></textarea>
        <textarea id="edb" placeholder="Encrypted Database location"
cols="100" rows="1"></textarea>
                <textarea id="output" placeholder="Output location"
cols="100" rows="1"></textarea>
        <button type="button" id="button">Search!</button>
    </body>
    </html>
```

APÊNDICE D – Código Fonte Cliente

PatientSpec.json

```
[
   {
      "name": "GenerateQuery",
      "params": {
         "preferences": {}
      },
      "returns" : ""
   },
   {
      "name": "CheckResult",
      "params": {
         "preferences": {},
         "result_path": ""
      },
      "returns" : ""
   }
]
```

AbstractStubPatientServer.h

```
        /**
         * This file is generated by jsonrpcstub, DO NOT CHANGE IT
MANUALLY!
         */

        #ifndef
JSONRPC_CPP_STUB_ABSTRACTSTUBPATIENTSERVER_H_
        #define
JSONRPC_CPP_STUB_ABSTRACTSTUBPATIENTSERVER_H_

        #include <jsonrpccpp/server.h>

        class AbstractStubPatientServer : public
jsonrpc::AbstractServer<AbstractStubPatientServer>
        {
          public:

AbstractStubPatientServer(jsonrpc::AbstractServerConnector &conn,
jsonrpc::serverVersion_t type = jsonrpc::JSONRPC_SERVER_V2) :
jsonrpc::AbstractServer<AbstractStubPatientServer>(conn, type)
            {

this->bindAndAddMethod(jsonrpc::Procedure("GenerateQuery",
jsonrpc::PARAMS_BY_NAME, jsonrpc::JSON_STRING,
"preferences",jsonrpc::JSON_OBJECT, NULL),
&AbstractStubPatientServer::GenerateQueryI);

this->bindAndAddMethod(jsonrpc::Procedure("CheckResult",
jsonrpc::PARAMS_BY_NAME, jsonrpc::JSON_STRING,
"preferences",jsonrpc::JSON_OBJECT,"result_path",jsonrpc::JSON_ST
RING, NULL), &AbstractStubPatientServer::CheckResultI);
            }

                inline virtual void GenerateQueryI(const Json::Value
&request, Json::Value &response)
            {
               response = this->GenerateQuery(request["preferences"]);
            }
```

```
                    inline virtual void CheckResultI(const Json::Value
&request, Json::Value &response)
            {
                    response = this->CheckResult(request["preferences"],
request["result_path"].asString());
            }
                virtual std::string GenerateQuery(const Json::Value&
preferences) = 0;
                    virtual std::string CheckResult(const Json::Value&
preferences, const std::string& result_path) = 0;
        };

        #endif
//JSONRPC_CPP_STUB_ABSTRACTSTUBPATIENTSERVER_H_

        PatientServer.cpp
        #include "stub/AbstractStubPatientServer.h"
        #include <jsonrpccpp/server/connectors/httpserver.h>
        #include <iostream>
        #include <stdio.h>
        #include "sse/Client.hpp"

        using namespace jsonrpc;
        using namespace std;

        class PatientServer : public AbstractStubPatientServer
        {
           public:
              PatientServer(AbstractServerConnector &connector);

                virtual std::string GenerateQuery(const Json::Value&
preferences);
                    virtual std::string CheckResult(const Json::Value&
preferences, const std::string& result_path);
        };

        PatientServer::PatientServer(AbstractServerConnector
&connector) :
```

```
        AbstractStubPatientServer(connector)
    {
    }

    std::string PatientServer::GenerateQuery(const Json::Value&
preferences)
    {
        Client *c = new Client();
        std::string stag("ACTION:GET");
        std::string xtags = "";

        Json::Value allow = preferences["allow"];
        std::vector<std::string> keys = allow.getMemberNames();
        for(auto iter = keys.begin(); iter != keys.end(); ++iter) {
            xtags += "OB:" + (*iter) + " ";
             for (Json::Value::ArrayIndex i = 0; i != allow[(*iter)].size();
i++) {
                xtags+= "ACTOR:" + allow[(*iter)][i].asString() + " ";
            }
        }
        std::string query = c->conjunctiveSearch(stag, xtags);
        return query;
    }

    std::string PatientServer::CheckResult(const Json::Value&
preferences, const std::string& result_path)
    {
        Client *c = new Client();
        std::vector<boost::dynamic_bitset<unsigned char>> result;
        result = c->fetchResult(result_path);

        //Algorithm to check the results
        Json::Value allow = preferences["allow"];
        std::vector<std::string> keys = allow.getMemberNames();
        auto result_iter = result.begin();
                boost::dynamic_bitset<unsigned char> violations
((*result_iter).size());
        for(auto iter = keys.begin(); iter != keys.end(); ++iter) {
```

```
            boost::dynamic_bitset<unsigned char> ob;
            ob = (*result_iter);
            boost::dynamic_bitset<unsigned char> actors (ob.size());
             for (Json::Value::ArrayIndex i = 0; i != allow[(*iter)].size();
i++) {
               result_iter++;
               boost::dynamic_bitset<unsigned char> tst = (*result_iter);
               actors |= tst;
            }
            violations |= ~(~(ob) | actors);
            result_iter++;
          }
          std::string result_rpc = "";
          int count = violations.count();
            result_rpc += std::to_string(count) + " entries violates your
privacy";
          if(count > 0) {
              std::string violations_str;
              boost::to_string(violations, violations_str);
              result_rpc += "\nIndexes (1 equals privacy breach): " +
                violations_str;

          }
          return result_rpc;
        }

        int main()
        {
          HttpServer httpserver(8585);
          PatientServer s(httpserver);
          s.StartListening();
          getchar();
          s.StopListening();
          return 0;
        }

        StubPatientClient.js
        /**
```

```
        * This file is generated by jsonrpcstub, DO NOT CHANGE IT
MANUALLY!
        */
       function Patient(url) {
          this.url = url;
          var id = 1;

             function doJsonRpcRequest(method, params, methodCall,
callback_success, callback_error) {
             var request = {};
             if (methodCall)
                request.id = id++;
             request.jsonrpc = "2.0";
             request.method = method;
             if (params !== null) {
                request.params = params;
             }
             JSON.stringify(request);

             $.ajax({
                type: "POST",
                url: url,
                data: JSON.stringify(request),
                success: function (response) {
                   if (methodCall) {
                                 if (response.hasOwnProperty("result") &&
response.hasOwnProperty("id")) {
                            callback_success(response.id, response.result);
                         } else if (response.hasOwnProperty("error")) {
                            if (callback_error != null)

callback_error(response.error.code,response.error.message);
                         } else {
                            if (callback_error != null)
                                     callback_error(-32001, "Invalid Server
response: " + response);
                         }
                      }
```

```
                },
                error: function () {
                    if (methodCall)
                        callback_error(-32002, "AJAX Error");
                },
                dataType: "json"
            });
            return id-1;
        }
            this.doRPC = function(method, params, methodCall,
callback_success, callback_error) {
             return doJsonRpcRequest(method, params, methodCall,
callback_success, callback_error);
        }
    }

    Patient.prototype.GenerateQuery = function(preferences,
callbackSuccess, callbackError) {
        var params = {preferences : preferences};
            return this.doRPC("GenerateQuery", params, true,
callbackSuccess, callbackError);
    };
    Patient.prototype.CheckResult = function(preferences,
result_path, callbackSuccess, callbackError) {
          var params = {preferences : preferences, result_path :
result_path};
             return this.doRPC("CheckResult", params, true,
callbackSuccess, callbackError);
    };

    patient.html
    <!DOCTYPE html>
    <html>
    <head>
    <meta                          content="text/html;charset=utf-8"
http-equiv="Content-Type">
    <title>Patient software</title>
```

```
        <script src="http://code.jquery.com/jquery-2.1.1.min.js"
type="text/javascript"></script>
        <!--
           This file needs to be generated using jsonrpcstub:
           cd build
                              ./bin/jsonrpcstub ../src/examples/spec.json
--js-client=StubClient --js-client-file=../src/examples/stubclient.js
        -->
        <script                                  src="js/StubPatientClient.js"
type="text/javascript"></script>
        <script type="text/javascript">
        $(document).ready(function () {
           function showResult(id, result) {
              alert(result);
           }
           function showError(code, message) {
              alert("Error: " + code + " -> " + message);
           }

           $('#query_button').click(function() {
              var patient = new Patient("http://localhost:8585");
               patient.GenerateQuery(JSON.parse($("#preferences").val())
,
                 function (id, result) {
                    $("#query").val(result);
                 }, showError);
           });

           $('#search_button').click(function() {
              var patient = new Patient("http://localhost:8585");
                 patient.CheckResult(JSON.parse($("#preferences").val()),
$('#result_path').val(),showResult, showError);
           });
        });
        function displayResult(result) {
           alert(JSON.stringify(result));
        }
        </script>
```

```
        </head>
        <body>
            <textarea id="preferences" placeholder="Preferences JSON"
cols="50" rows="15"></textarea>
                        <button type="button" id="query_button">Gen
Query!</button>
             <textarea id="result_path" placeholder="Result query path"
cols="60" rows="1"></textarea>
            <button type="button" id="search_button">Search!</button>
            <br>
                <textarea id="query" placeholder="Query" cols="100"
rows="30"></textarea>
        </body>
        </html>
```

APÊNDICE E – Código Fonte Sistema Médico

MedicalSystemSpec.json

```
[
   {
      "name": "GetPatientPreferences",
      "returns" : {}
   },
   {
      "name": "GenerateEncryptedLogs",
      "params": {
         "output_path": ""
      },
      "returns" : false
   }
]
```

AbstractStubMedicalSystemServer.h

```
/**
 * This file is generated by jsonrpcstub, DO NOT CHANGE IT MANUALLY!
 */

#ifndef JSONRPC_CPP_STUB_ABSTRACTSTUBMEDICALSYSTEMSERVER_H_
#define JSONRPC_CPP_STUB_ABSTRACTSTUBMEDICALSYSTEMSERVER_H_

#include <jsonrpccpp/server.h>

class AbstractStubMedicalSystemServer : public jsonrpc::AbstractServer<AbstractStubMedicalSystemServer>
{
   public:
```

```
AbstractStubMedicalSystemServer(jsonrpc::AbstractServerConnector
&conn, jsonrpc::serverVersion_t type =
jsonrpc::JSONRPC_SERVER_V2) :
jsonrpc::AbstractServer<AbstractStubMedicalSystemServer>(conn,
type)
            {

this->bindAndAddMethod(jsonrpc::Procedure("GetPatientPreferences",
jsonrpc::PARAMS_BY_NAME, jsonrpc::JSON_OBJECT, NULL),
&AbstractStubMedicalSystemServer::GetPatientPreferencesI);

this->bindAndAddMethod(jsonrpc::Procedure("GenerateEncryptedLogs
", jsonrpc::PARAMS_BY_NAME, jsonrpc::JSON_BOOLEAN,
"output_path",jsonrpc::JSON_STRING, NULL),
&AbstractStubMedicalSystemServer::GenerateEncryptedLogsI);
            }

             inline virtual void GetPatientPreferencesI(const Json::Value
&request, Json::Value &response)
            {
               (void)request;
               response = this->GetPatientPreferences();
            }
                    inline virtual void GenerateEncryptedLogsI(const
Json::Value &request, Json::Value &response)
            {
                                                    response =
this->GenerateEncryptedLogs(request["output_path"].asString());
            }
            virtual Json::Value GetPatientPreferences() = 0;
                 virtual bool GenerateEncryptedLogs(const std::string&
output_path) = 0;
        };

        #endif
//JSONRPC_CPP_STUB_ABSTRACTSTUBMEDICALSYSTEMSER
VER_H_
```

MedicalSystemServer.cpp

```
#include "stub/AbstractStubMedicalSystemServer.h"
#include <jsonrpccpp/server/connectors/httpserver.h>
#include <iostream>
#include <stdio.h>
#include <fstream>
#include "sse/Client.hpp"

using namespace jsonrpc;
using namespace std;

class MedicalSystemServer : public AbstractStubMedicalSystemServer
{
  public:
                MedicalSystemServer(AbstractServerConnector &connector);

      virtual Json::Value GetPatientPreferences();
          virtual bool GenerateEncryptedLogs(const std::string& output_path);
};

MedicalSystemServer::MedicalSystemServer(AbstractServerConnector &connector) :
  AbstractStubMedicalSystemServer(connector)
{
}

Json::Value MedicalSystemServer::GetPatientPreferences()
{
  Json::Value return_value;
                                                    std::ifstream file("/home/luz/tcc/libssebitmap/medsystem/preferences.json");
  file >> return_value;
  return return_value;
}
```

```
bool MedicalSystemServer::GenerateEncryptedLogs(const std::string& output_path)
{
   Client *c = new Client();

c->setupDB("/home/luz/tcc/libssebitmap/resources/datasets/logs", 30, output_path);
   return true;

}
int main()
{
   HttpServer httpserver(8484);
   MedicalSystemServer s(httpserver);
   s.StartListening();
   getchar();
   s.StopListening();
   return 0;
}

StubMedicalSystemClient.js
/**
 * This file is generated by jsonrpcstub, DO NOT CHANGE IT MANUALLY!
 */
function MedicalSystem(url) {
   this.url = url;
   var id = 1;

      function doJsonRpcRequest(method, params, methodCall, callback_success, callback_error) {
         var request = {};
         if (methodCall)
            request.id = id++;
         request.jsonrpc = "2.0";
         request.method = method;
         if (params !== null) {
```

```
            request.params = params;
        }
        JSON.stringify(request);

        $.ajax({
            type: "POST",
            url: url,
            data: JSON.stringify(request),
            success: function (response) {
                if (methodCall) {
                        if (response.hasOwnProperty("result") &&
response.hasOwnProperty("id")) {
                    callback_success(response.id, response.result);
                } else if (response.hasOwnProperty("error")) {
                    if (callback_error != null)

callback_error(response.error.code,response.error.message);
                } else {
                    if (callback_error != null)
                            callback_error(-32001, "Invalid Server
response: " + response);
                }
                }
            },
            error: function () {
                if (methodCall)
                    callback_error(-32002, "AJAX Error");
            },
            dataType: "json"
        });
        return id-1;
    }
            this.doRPC = function(method, params, methodCall,
callback_success, callback_error) {
             return doJsonRpcRequest(method, params, methodCall,
callback_success, callback_error);
    }
}
```

```
        MedicalSystem.prototype.GetPatientPreferences =
function(callbackSuccess, callbackError) {
            var params = null;
                return this.doRPC("GetPatientPreferences", params, true,
callbackSuccess, callbackError);
        };
        MedicalSystem.prototype.GenerateEncryptedLogs =
function(output_path, callbackSuccess, callbackError) {
            var params = {output_path : output_path};
              return this.doRPC("GenerateEncryptedLogs", params, true,
callbackSuccess, callbackError);
        };

        medicalsystem.html
        <!DOCTYPE html>
        <html>
        <head>
        <meta content="text/html;charset=utf-8"
http-equiv="Content-Type">
        <title>Medical System</title>
        <script src="http://code.jquery.com/jquery-2.1.1.min.js"
type="text/javascript"></script>
        <script src="js/StubMedicalSystemClient.js"
type="text/javascript"></script>
        <script type="text/javascript">
        $(document).ready(function () {
            function showResult(id, result) {
                alert("ID: " + id + " Result: " + result);
            }
            function showError(code, message) {
                alert("Error: " + code + " -> " + message);
            }


            $('#preferences_button').click(function() {
                                    var medicalsystem = new
MedicalSystem("http://localhost:8484");
```

```
            medicalsystem.GetPatientPreferences(function (id, result) {
                $("#preferences").val(JSON.stringify(result));
            }, showError);
        });
        $('#edb_button').click(function() {
                                    var medicalsystem = new
MedicalSystem("http://localhost:8484");
            medicalsystem.GenerateEncryptedLogs($("#output").val(),
showResult, showError);
        });

    });
    function displayResult(result) {
        alert(JSON.stringify(result));
    }
    </script>
    </head>
    <body>
      <textarea id="preferences" placeholder="Preferences JSON"
cols="50" rows="15"></textarea>
          <button type="button" id="preferences_button">Get
Pref!</button>
      <textarea id="output" placeholder="Output location" cols="60"
rows="1"></textarea>
      <button type="button" id="edb_button">Gen EDB!</button>
    </body>
    </html>
```

Privacidade e verificabilidade de acesso em sistemas médicos usando *searchable encryption*

Gabriel Luz[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Campus Universitário – Florianópolis – SC – Brazil

gabriel.luz@grad.ufsc.br

Abstract. *Medical systems have risen in popularity because they make the data sharing between patients and the medical staff easier. However, the patient must be in control of this data sharing. The patient must be able to verify the access to his data to confirm if there has been any violation to his privacy preferences. It is desirable that the patient can send the log entries to his data and his data sharing preferences to a third-party that could verify if there has been any breach to his privacy. However, it is possible to extract a lot of personal information given the access logs to someone's medical data such as medical exams and doctor visits. Some solutions on the state of the art use searchable encryption schemes to solve this issue. In this work, one of the proposed solutions was be implemented along with a prototype of a medical system using this solution to simulate a real life scenario and verify its applicability.*

Resumo. *Sistemas médicos têm se tornado populares pois facilitam o compartilhamento de dados entre pacientes e médicos. Entretanto, o compartilhamento deve estar sob controle do paciente, que irá decidir que informações devem ser compartilhadas e quem deve ter acesso a elas. O paciente deve ter o poder de verificar os acessos que foram feitos a seus dados, podendo assim confirmar se houve algum acesso que não esteja de acordo as regras estabelecidas. O paciente pode não ter o conhecimento técnico necessário para fazer esta verificação. Logo, é desejável também que o paciente possa enviar os registros dos acessos para um verificador, juntamente com suas preferências de acesso, para que possa ser verificado se houve alguma quebra das regras de acesso definidas pelo paciente. Dessa maneira, o processo de verificação seria facilitado pelo verificador, aumentando as chances dos pacientes utilizarem e realizarem essa verificação frequentemente. Entretanto, o uso de um verificador pode comprometer a privacidade dos dados pois é possível obter diversas informações somente com os registros de acesso, como qual médico foi visitado ou quais exames foram realizados. Portanto, ao permitir que outra pessoa acesse estes registros, o sigilo das informações do paciente pode estar comprometido. Soluções no estado da arte utilizam técnicas de searchable encryption para solucionar este problema. Este trabalho visa implementar um modelo proposto no estado da arte em um protótipo de sistema médico e avaliar a aplicabilidade do modelo.*

1. Introdução

Quando o programa de espionagem dos Estados Unidos veio a conhecimento público pelas denúncias realizadas por Edward Snowden, o tópico da privacidade de dados tornou-se extremamente popular (BBC NEWS, 2016). A variedade de dados contendo informações pessoais tem crescido exponencialmente nas últimas décadas (SWEENEY, 2002). Apesar de existir uma grande discussão no campo da privacidade sobre o que deve ser considerado sigiloso e o que deve ser aberto ao público, ainda é um consenso que dados médicos de pacientes são dados sigilosos, que só poderão ser compartilhados sob o consentimento do paciente. Existem várias soluções apresentadas no estado da arte para resolver problemas relacionados à privacidade e verificabilidade usando *searchable encryption*.

Searchable encryption é um tipo de criptografia que permite que terceiros realizem uma consulta no dado cifrado, satisfazendo uma condição sem ter que decifrar o texto. Para isso, alguns esquemas de criptografia consultável requerem que se tenha uma palavra-chave para ser usada como um *search token* para cada consulta, vinculando uma palavra-chave para cada dado para que seja possível fazer uma pesquisa somente com as palavras-chave *(search tokens)*. Logo, para esquemas que usam *search tokens*, para cada consulta é necessária a criação de um *search token* por quem possuir a chave e o envio do *token* para quem realizará a operação de consulta.

A verificação de acesso é comumente requerida em uma vasta área de tecnologias, principalmente quando se trata do acesso a dados médicos, onde o sigilo não pode ser quebrado. Aplicações que requerem a verificabilidade encontram problemas para garantir a confiabilidade dos dados e também para dar a sensação de segurança aos usuários de que seus dados estão sendo armazenados com sigilo.

O modelo proposto por Idalino, Spagnuelo e Martina (2017) usa conceitos de verificabilidade e criptografia consultável para solucionar problemas relacionados à verificabilidade de acessos em sistemas médicos. O *framework* proposto possibilita que o paciente verifique se houve algum acesso não autorizado aos seus dados e também possibilita que este autorize um terceiro a fazer essa verificação, enviando os registros de acesso a essa pessoa. Este processo é feito sem que esta pessoa que fará a verificação consiga obter qualquer informação a partir dos registros de acesso.

2. *Searchable Encryption*

O termo utilizado na literatura é *Searchable Encryption*, porém, o autor pode se referenciar a *searchable encryption* como criptografia consultável como tradução direta ao longo do texto. A criptografia consultável permite que um servidor faça buscas em documentos sem que possa extrair qualquer informação destes documentos (ABDELRAHEEM, 2016).

Alguns esquemas de criptografia consultável possibilitam fazer pesquisas no ciphertext enquanto a maioria dos esquemas permite a geração de um index com keywords que são utilizadas para a geração da query e a pesquisa posteriormente (BÖSCH et al., 2015). Para pesquisar, o cliente gera uma query com os predicados que deseja verificar e envia para o servidor, que por sua vez itera sobre os index e retorna os documentos que satisfazem o predicado. Os tipos de criptografia consultável e suas vantagens e desvantagens serão discutidas na seção 4.2 na escolha e implementação da criptografia consultável

para este trabalho. A Figura 1 ilustra um cenário client-server utilizando *searchable encryption*. O primeiro passo é o cliente gerar os índices cifrados para serem mandados à *cloud* junto com os dados cifrados.

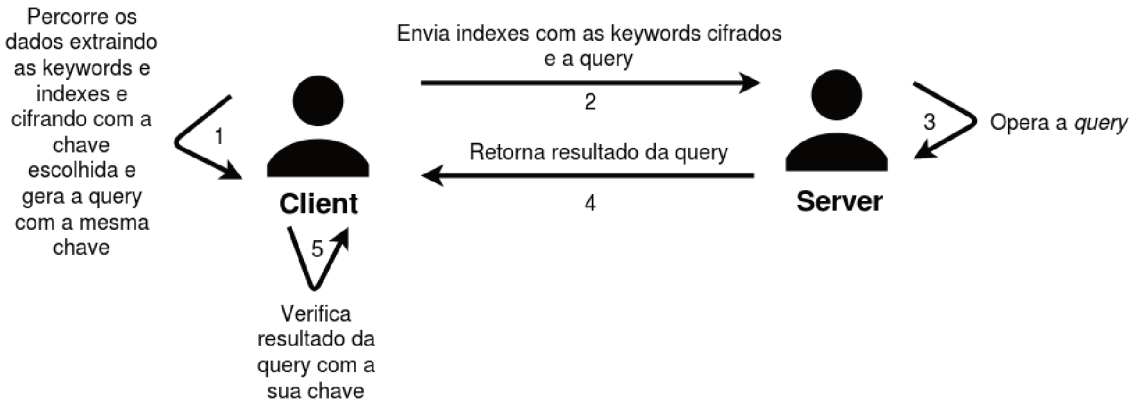

Figure 1. Searchable Encryption em um cenário client-cloud

3. Sistemas Médicos

Sistemas médicos é um termo genérico que abrange diferentes tipos de sistemas com diversas funcionalidades e objetivos. Uma meta em comum entre todos eles é a coleta e armazenamento de informações da saúde de um paciente em um formato digital. Essas informações podem ser disponibilizadas para o paciente ou a equipe médica. Essas informações incluem dados como histórico médico, dados demográficos, medicamentos, alergias, testes de laboratório, imagens de radiografia entre outras informações. O sistema também pode conter informações adicionadas pelo paciente como por exemplo seu peso por meio de uma balança digital conectada a seu celular e outras informações manualmente adicionadas pelo paciente como humor ou dor. A grande vantagem de um sistema médico é o poder de busca que ele oferece para a equipe de saúde e a segurança de manter os dados armazenados digitalmente. Sem um sistema médico, o hospital tem que manter uma organização física de informações relacionadas a seus pacientes, o paciente também deve manter uma cópia de seus exames e quando realizar uma visita ao hospital deve procurar todos os exames relevantes e levar consigo. Em muitos casos esses exames podem ser perdidos ou o paciente pode não levar exames relevantes para a visita. O uso de sistemas médicos soluciona esses problemas de perda e replicação de dados e possibilita a equipe médica encontrar tendências no histórico unificado do paciente. Além disso, se o sistema médico for amplamente adotado, principalmente em instituições públicas atingindo uma grande parte da população, possibilita o estudo estatístico muito preciso.

Este trabalho terá foco em sistemas médicos orientados aos pacientes. São sistemas onde o paciente tem poder sobre seus dados. As definições de um sistema de saúde com essa filosofia começaram a aparecer no começo dos anos 2000. No começo eram definidos sistemas onde o paciente mantém as informações no sistema atualizadas, adicionando seus resultados de exame, diagnósticos, informações geradas pelo paciente entre outras informações mencionadas anteriormente. Além de armazenar as informações, algumas implementações do sistema também possibilita a troca de mensagens entre o paciente e o médico, marcar consultas e exames e lembretes. Eliminando essas barreiras

de comunicação e possibilitando a troca de informações facilmente, sistemas médicos poupam o tempo gasto com consultas e podem salvar vidas auxiliando a equipe médica a diagnosticar problemas de saúde precocemente ou receitar um tratamento eficiente para a saúde do paciente baseado nos resultados dos tratamentos anteriores.

Antes do surgimento de sistemas médicos existiam outras maneiras de manter um sistema por meio de diferentes plataformas que conseguiam atingir algumas das funcionalidades dos sistemas médicos atuais. A mais básica é manter arquivos impressos em papéis e ter uma organização de arquivos, para compartilhar com o médico tirar cópias dos documentos necessários e levar para a visita. O armazenamento em papel consegue atingir a funcionalidade de organização dos dados como um sistema médico, porém depende do usuário o nível de organização e tem problemas com perda de dados em caso de algum desastre, ou esquecimento do usuário em levar outros exames necessários. Além dessa plataforma, era comum manter o sistema médico com dispositivos eletrônicos, porém localmente. Pode ser utilizado um software específico de sistemas médicos, onde o usuário mantém uma cópia dos dados local que podem estar cifrados ou não, e o usuário deve combinar um meio de comunicação para compartilhar os dados com o seu médico. O problema com este meio é a falta de um padrão para compartilhamento com os funcionários da saúde e o usuário ser responsável pelos backups de seus dados. Por último, soluções modernas que gerenciam o compartilhamento, recuperação de dados, adição de informações automática. Esse tipo de plataforma promove os pontos positivos anteriormente citados, porém gera algumas questões controversas. Uma das questões é como essa tecnologia pode ameaçar a privacidade dos usuários. O medo dos usuários em ter todos os seus dados médicos online pode atrasar a adoção dos sistemas médicos. Soluções que promovem a garantia de confiabilidade e maior segurança para os pacientes em relação aos registros de acesso, como a apresentada neste trabalho, podem transformar a expressividade da adoção dos sistemas médicos.

4. Modelo

O modelo proposto por Idalino, Spagnuelo e Martina (2017) tem como objetivo promover o poder aos usuários de sistemas médicos de verificar a concordância dos acessos ao sistema médico com suas preferências de acesso à seus dados médicos. Além disso, o usuário também tem o poder de verificar com um terceiro, onde o verificador não poderá extrair informações a partir dos logs de acesso. Cada paciente tem sua política de privacidade, contendo o que cada membro da equipe médico pode acessar de seus dados.

Os autores assumem três diferentes participantes nesse sistema, são eles: Sistema médico: Armazena os dados médico dos pacientes e é responsável por compartilhar os dados entre os participantes e fornecer uma lista de logs com os acessos aos dados do paciente Paciente: Usuário do sistema, dono das informações contidas no sistema médico. Pode pedir acesso aos logs do sistema médico para verificar por si mesmo ou enviar para qualquer terceiros para fazerem a verificação. Verificador: Terceiro que pode fornecer serviços para verificação de acesso a dados de pacientes sem conseguir obter informações sobre o paciente.

Além disso, os autores também definem requerimentos para o sistema como: Verificação Automatizada: O sistema médico deve fornecer meios para facilitar a verificação para o paciente. Auditoria de dados independente: Um terceiro pode veri-

ficar os dados de acesso aos dados do paciente. Privacidade: Em nenhum momento a privacidade do paciente deve ser violada.

Para isso, foi utilizado criptografia consultável, que permite fazer buscas no dado cifrado sem revelar informações sobre os dados. Os logs de registro de acesso precisam passar pelo algoritmo de criptografia consultável utilizado, gerando os índices e as keywords de cada índice. O paciente e o sistema médico fazem um acordo de uma chave simétrica, que é usada para cifrar os índices e as keywords. O paciente, por sua vez, com posse da chave em que foi cifrado os índices e as keywords, deve criar uma query de acordo com as suas preferências de acesso. Então, ele pode enviar os dados cifrados para o verificador junto com a query, que pode auditar esses logs cifrados e retornar o resultado afirmando se todos os logs respeitam as preferências de acesso ou não, sem obter alguma informação sobre o paciente. A Figura 2 ilustra o modelo proposto.

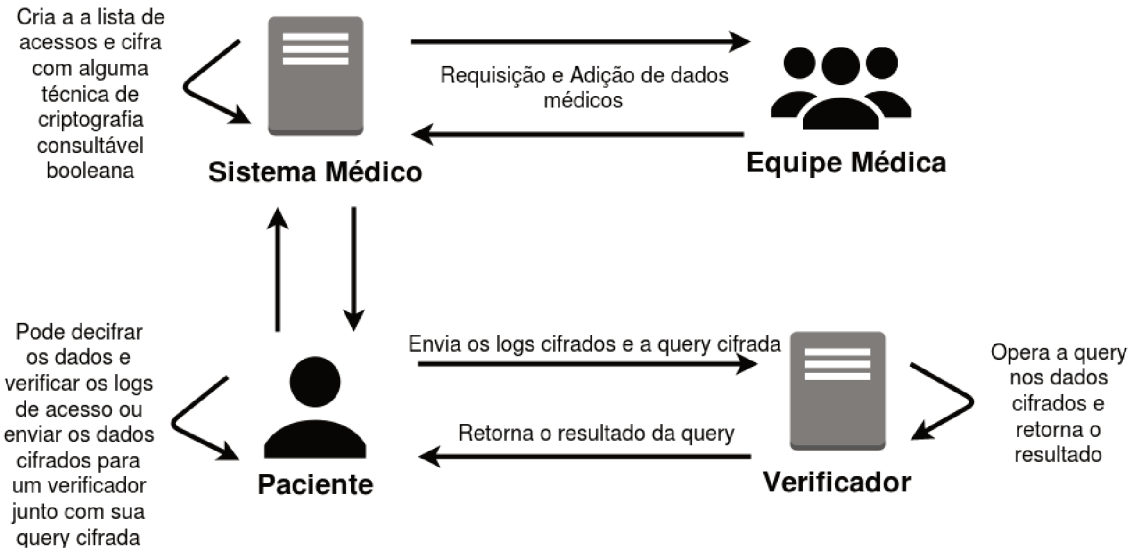

Figure 2. Modelo

O modelo deles assume o sistema médico como honesto mas não confiável. Desse modo, o sistema não é malicioso, mas pode ser abusado, como alguém acessar algum dado médico que não poderia. Eles consideram o verificador honesto mas curioso, se tiver alguma informação sobre o paciente disponível ele não respeitaria o sigilo. Eles também consideram a existência de um atacante que pode reter qualquer informação disponível ao verificador.

Para evitar colusão entre o sistema médico e o verificador, os autores recomendam a implementação de verificadores por diferentes entidades. Dessa forma, colusões seriam evitadas, pois a possibilidade de um paciente utilizar outro verificador honesto é muito alta e o sistema médico perderia completamente a sua credibilidade, o que é o objetivo oposto da utilização deste trabalho. Além disso, verificadores podem ser verificados enviando uma requisição de verificação onde o resultado é conhecido e comparando com a resposta do verificador.

O trabalho proposto pelos autores não leva em consideração a integridade dos logs de acesso por isso ser de responsabilidade do sistema médico. Eles assumem que o sistema médico mantém logs seguros e íntegros.

4.1. Algoritmos Utilizados pelo Modelo

O algoritmo descrito na Figura 3 é o qual o sistema médico gera os logs de acesso cifrados. O sistema médico cifra cada log individualmente, criando um index para cada um, contendo as palavras-chave do log, como a ação e o autor dessa ação. A geração do index depende do algoritmo de criptografia consultável usado.

Algorithm 1 EncryptLogs($logs[n]$)

Input: array of $logs$ with n entries
for each $i \in \{1, \ldots, n\}$ **do**
  $c[i] = \text{Enc}(k, logs[i])$
  $\text{keywords} = \text{extractKeywords}(logs[i])$
  $\text{index}[i] = \text{generateIndex}(k, \text{keywords})$
end for
Output: c, index

Figure 3. Algoritmo EncryptLogs retirado de Idalino, Spagnuelo e Martina (2017)

Depois de gerar e cifrar os índices com as keywords de cada log, eles são enviados ao paciente. Ele, por sua vez, pode escolher analisar os logs por sí mesmo ou dar essa tarefa a um verificador. Caso decida enviar para um verificador, é necessário explicar ao verificador o que ele deve buscar nesses logs. Para isso, o paciente gera uma query. A Figura 4 mostra esse processo da geração da query. A query é gerada de acordo com a política de privacidade acordada com o sistema médico. Essa política de privacidade define o que cada membro da equipe médica pode acessar nos dados do paciente. Se a política de privacidade não for alterada, o paciente pode reutilizar essa query futuramente. A query é representada como uma expressão booleana na disjunctive normal form (DNF), que é uma normalização de uma disjunção de cláusulas conjuntivas, utilizando um id identificando a pessoa e um action identificando a ação que essa pessoa pode realizar. Uma query ficaria no seguinte formato (id1 / action1) (id1 / action2) (id2 / action1). É utilizado a DNF pois com ela é possível realizar a disjunção de cláusulas conjuntivas, nesse caso é a disjunção das conjunções dos ids dos membros médicos com os actions que eles podem realizar. Logo, é uma query que verifica se um log está de acordo com a política de privacidade.

Depois de gerar a query, o paciente deve enviar os logs cifrados e a query para o verificador. Isso é suficiente para o verificador fazer a busca nos logs e retornar os que estão de acordo com a query. A busca por parte do verificador é mostrada na Figura 5. Note que este algoritmo também depende do esquema de criptografia consultável booleana utilizado. Depois de realizar a verificação, o verificador envia para o paciente uma lista de logs cifrados que estão de acordo com a query.

O resultado pode ser simplificado mandando uma mensagem positiva ou negativa se há violação da política de acesso do sistema médico a partir do resultado da query a partir da condição de que todos os índices estão de acordo com a query. O algoritmo 2 de geração da query também pode ser facilmente modificado para indicar quais logs violam a política e o paciente pode decifrar esses logs para descobrir qual foi a violação. Em

Algorithm 2 GenerateQuery(π)

Input: Policy $\pi = \{(id_i, action_i)\}$ with m clauses
DNF = empty string
for each $i \in \{1, \ldots, m\}$ **do**
  DNF = DNF $\|(id_i \wedge action_i)$
  if $i \neq m$ **then**
    DNF = DNF $\|\vee$
  end if
end for
$\mathcal{Q}$ = generateQuery(DNF)
Output: $\mathcal{Q}$

Figure 4. Algoritmo GenerateQuery retirado de Idalino, Spagnuelo e Martina (2017)

Algorithm 3 Search($\mathcal{Q}, c[n], index[n]$)

Input: Encrypted query $\mathcal{Q}$, vector c with n encrypted logs, and their indexes
for each $i \in \{1, \ldots, n\}$ **do**
  $r = \text{test}(\mathcal{Q}, index[i])$
  if $r =$ **true then**
    $result$.add($c[i]$)
  end if
end for
Output: $result$

Figure 5. Algoritmo Search retirado de Idalino, Spagnuelo e Martina (2017)

nenhum momento o verificador obtém qualquer dado sobre os logs de acesso, ou sobre a query. A única informação para ele é a quantidade de logs que estão de acordo com a query enviada, que não tem seus termos em texto plano para o verificador.

5. Implementação

5.1. Implementação de um algoritmo de searchable encryption

Para a implementação do algoritmo de criptografia consultável foi escolhido o algoritmo proposto por Abdelraheem et al. (2016). Os autores propuseram um algoritmo de criptografia consultável que permite operações booleanas. Apesar de que este algoritmo não foi diretamente mencionado no trabalho proposto por Idalino, Spagnuelo e Martina (2017), este trabalho é uma melhoria do trabalho proposto por Cash et al. (2013) citado pelos autores. Ele possui as características necessárias apontadas pelos autores para ser utilizado como o algoritmo de criptografia consultável neste modelo. A mais importante sendo a de utilização de query com múltiplas keywords. Além disso, Abdelraheem et al. (2016) disponibilizou os código dos testes realizados em sua pesquisa, o que facilitou a implementação da biblioteca OpenSSE.

Com isso, a implementação da criptografia consultável foi a adaptação do código de algoritmo de criptografía consultável disponibilizado por Abdelraheem et al. (2016) em uma pequena biblioteca chamada OpenSSE. Ela possui duas classes Client e Server.

O Client é responsável por coletar as keywords e indexes e cifrar utilizando uma chave escolhida. O Client também é responsável pela criação da query e verificação do resultado. O Server é realiza a consulta nos indexes e keywords cifrados.

O código disponibilizado por Abdelraheem et al. (2016) foi feito para realizar os testes de desempenho disponibilizados em seu trabalho, as funções do Client e Server se comunicavam entre si via sockets. Com isso, a primeira alteração necessária foi abstrair os algoritmos de criptografia consultável para que funcionassem independente da conexão do Client com o Server.

Os algoritmos de criptografia consultável clássicos só permitem a consulta de uma única keyword, porém muitos problemas, como o apresentado neste trabalho, necessitam de um algoritmo mais expressivo onde é possível fazer queries booleanas. Por exemplo, com um algoritmo clássico é possível fazer a query dos logs de dados médicos onde a ob é igual a 'X-RAY'. Porém, no escopo deste trabalho é necessário o uso de queries mais expressivas como por exemplo, uma consulta onde ob é igual a 'X-RAY' e actor é igual a '1'. De acordo com Kamara e Moataz (2017) uma solução ingênua de realizar essa consulta denominada de response-revealing seria fazer a pesquisa onde ob é igual a 'X-RAY', outra pesquisa onde id é igual a '1' e realizar a intersecção entre os dois resultados. O problema desta solução ingênua é que apesar de não vazar diretamente as keywords, o server teria os índices da primeira e da segunda consulta. Alguns tipos de ataques poderiam descobrir informações sensíveis sobre os dados cifrados com um número significante de queries. Com um algoritmo que suporta queries mais expressivas o servidor teria somente os índices da intersecção, ou seja, o equivalente a uma pesquisa de uma única keyword. Outra solução ingênua denominada de response-hiding por Kamara e Moataz (2017) são esquemas que o server não consegue descobrir os índices que são somente calculados no cliente, o problema de soluções como esta, de acordo com os autores, são que são ineficientes por passar informações redundantes. No caso do exemplo anterior, o problema de informações redundantes seria que o servidor responderia todos os índices onde ob é igual a 'X-RAY' e todos os índices onde actor é igual a '1' e o cliente resolveria a intersecção. Ainda de acordo com Kamara e Moataz (2017), qualquer esquema de criptografia consultável deve resolver um desses dois problemas das soluções ingênuas.

No trabalho desenvolvido, por utilizar um número relativamente pequeno de dados, não tem problema com a complexidade dos algoritmos, somente com a confidencialidade. Os algoritmos de criptografia consultável propostos por Cash et al. (2013) e Abdelraheem et al. (2016) são soluções baseadas em response-hiding. A queries nestes esquemas são feitas na forma w1 AND f (w2, w3, ..., wn). Os autores definem dois grupos, s-tag e x-tags. O grupo s-tag seria o w1 e os elementos do grupo das x-tags seriam as keywords w2,w3,wn. Neste algoritmo, o servidor realiza uma consulta de única palavra com o w1 e obtém os índices, a partir disso ele faz uma consulta com cada x-tag somente nos índices obtidos pelo w1. O server envia para o client os índices das x-tags cifrados, ou seja, o único conhecimento do server é a da query de w1, como uma consulta de palavra única. Após receber os resultados, o client pode decifrar os índices das x-tags e realizar as operações booleanas, no caso do trabalho dos autores seria w1 AND w2 AND w3 AND wn. O problema de informações redundantes neste algoritmo proposto é feito pela conjunção lógica entre a s-tag e as x-tags. Para obter um melhor desempenho teria

que se escolher uma s-tag com a menor frequência para economizar operações para as x-tags.

Este trabalho visava utilizar a criptografia consultável para realizar consultas no formato ((w1 AND w2) OR (w3 AND w4) OR (wn AND wn)) onde cada conjunção é entre o id e action. Dessa forma, não é possível utilizar o mesmo formato apresentado no trabalho de Abdelraheem et al. (2016). Para isso, foi adaptado para que o resultado das x-tags seja resolvido depois de ter chamado a classe Client para resolver o resultado onde é possível realizar qualquer operação booleana entre as x-tags. Com esta utilização de esquema de criptografia consultável com response-hiding, como alguns dos propostos por Idalino, Spagnuelo e Martina (2017), é necessário que o paciente opere sobre o resultado do verificador, como pode ser visto no novo modelo na Figura 6. Já que a query para cada log não é composta por múltiplos termos, e sim a possibilidade de fazer mais de uma pesquisa de única palavra sem vazamento de informações, precisa ser alterada a forma de procurar por violações, sem o uso da DNF, esse novo algoritmo será apresentado na implementação do sistema do Paciente.

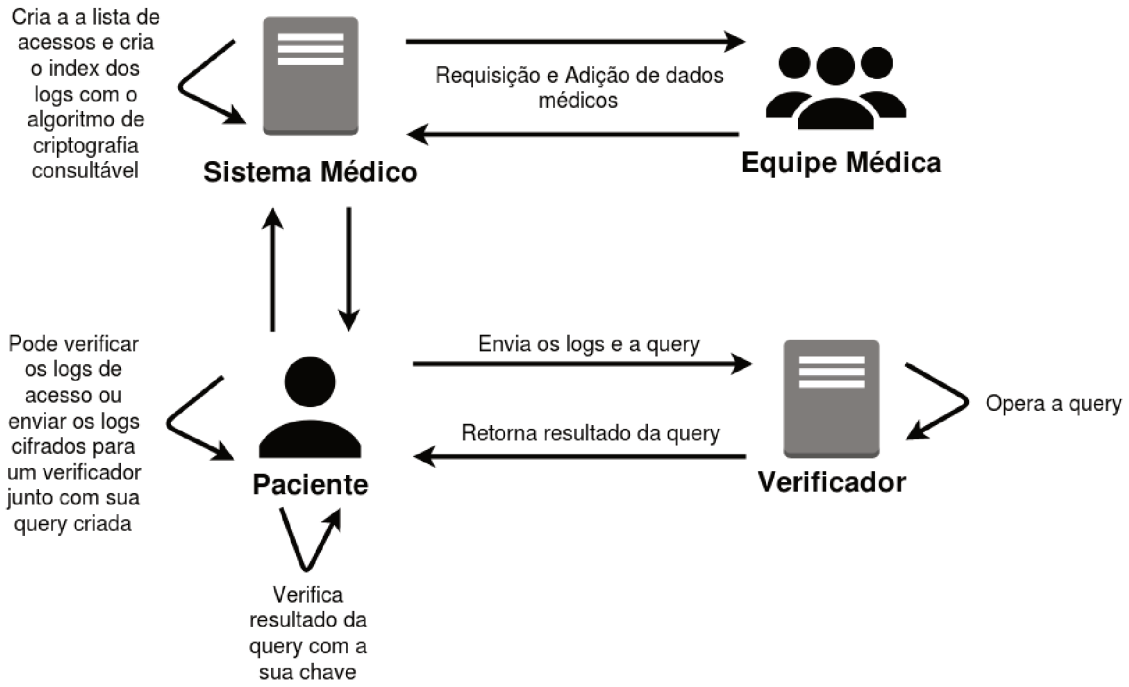

Figure 6. Modelo adaptado para *searchable encryption response-hiding*

A Classe Client possui 4 métodos importantes, são eles: setupDB, singleSearch, conjunctiveSearch e fetchResult. O método setupDB realiza a leitura dos arquivos e cria a lista de indexes e keywords análogo ao algoritmo descrito na Figura 3. Este método recebe como parâmetro a localização dos arquivos, o limite de arquivos para leitura e onde ele deve escrever o arquivo de saída dos indexes e keywords cifrados. Para facilitar a implementação, todas as funções utilizam uma chave que está hard-coded. O método setupDB antes das alterações realizadas, abria um socket para mandar as informações dos indexes e keywords para o cliente.

O método conjunctiveSearch da classe Client cria a query para busca com

múltiplas keywords, ela recebe como parâmetro a s-tag e as x-tags. Com a chave que está hardcoded ela cifra a query e retorna a query cifrada. Antes das alterações realizadas, este método abria um socket com o Server e mandava a query cifrada.

O método fetchResult da classe Client retorna um vetor de bitsets contendo os indexes que contém a x-tag a partir do resultado do Server. Esse método recebe como parâmetro o resultado da operação da query conjunctiveSearch no Server. Com isso, ela decifra cada resultado das x-tags e adiciona o bitset contendo os índices que contém a x-tag a um vetor. Antes das alterações, o método fetchResult era um método privado da classe, chamado no método conjunctiveSearch para decifrar o resultado e então realizar operações lógicas AND com todas as x-tags.

A classe Server possui três métodos importantes, são eles: setupEDB, singleKeywordSearch e conjunctiveSearch. O método setupEDB foi criado para que o objeto da classe Server inicializar suas s-tag e x-tags para as buscas a partir de um arquivo, ele recebe como parâmetro a localização do arquivo contendo os indexes e keywords cifrados. Após executar o setupEDB com um objeto da classe Server é possível chamar os métodos singleKeywordSearch e conjunctiveSearch. O método conjunctiveSearch recebe como parâmetro a query e a localização para o arquivo de saída. Ela primeiro executa uma singleKeywordSearch com a s-tag, e então para cada index que contém essa s-tag ele executa a query de cada x-tag e escreve o resultado no arquivo de saída.

5.2. Logs de Acesso

Os logs de acesso estão em um único arquivo. Eles são organizados na seguinte forma:

ACTION:GET DATE:2018-05-14 ACTOR:1 OB:X-RAY

Neste formato, o campo ACTION pode ser GET ou SUBMIT, representam o ato de pegar informações ou incluir informações sobre o paciente respectivamente. O campo DATE representa a data que foi realizada esta ação. O campo ACTOR representa o ator dessa ação, no caso alguém da equipe médica identificado pelo seu id numérico. O campo OB representa o artefato médico submisso a esta ação.

Para simplificar a implementação, todos os logs gerados para o trabalho são de um único paciente. Os logs são iterados pela função setupDB do Client na biblioteca OpenSSE.

Foram criados 10 ids para o campo ACTOR que representam 10 participantes da equipe médica. Para o campo OB foram utilizados alguns termos para dados de sistemas médicos. São eles: X-RAY, URINE-SAMPLE, DNA, MRI, LDL, HDL, VLDL, VHS, T3, T4. Totalizando 10 dados médicos, em sua maioria são resultados de exames de sangue para entrar no contexto de sistemas médicos.

5.3. Política de Acesso do Paciente

As preferências de acesso dos dados médicos dos usuários são definidas por um arquivo JSON que define a regra para cada tipo de dado médico com um array de ids que podem acessar estas informações de acordo com as preferências do usuário. As preferências de acesso usadas foram a descrita abaixo.

Desta maneira, o paciente “John Doe” permite, por exemplo, que os médicos com id de 1 e 4 podem acessar os resultados do exame de sangue t3 e t4. O paciente permite

```
{
    "patient": "John Doe",
    "allow": {
                "X-RAY": [1, 2, 3, 4],
                "URINE-SAMPLE": [2, 3],
                "DNA": [9],
                "MRI": [7],
                "LDL": [6],
                "HDL": [6],
                "VLDL": [6],
                "VHS": [1],
                "T3": [1,4],
                "T4": [1,4]
    }
}
```

Listing 1: Política de Acesso de John Doe

que várias pessoas acessem dados do X-RAY como os médicos com ids 1, 2, 3, 4.

5.4. Sistema Médico

O Sistema Médico mantém os logs de acesso e as preferências dos usuários. Ele tem duas funções, são elas: GetPatientPreferences, GenerateEncryptedLogs. No escopo deste protótipo, o GetPatientPreferneces retorna o JSON relacionado ao paciente John Doe. Para gerar os indexes e keywords cifrados, o sistema médico cria um objeto Server da biblioteca OpenSSE e pede para cifrar para um arquivo. O fornecimento dos logs pelo sistema médico está fora do escopo deste protótipo.

A função GenerateEncryptedLogs, no caso deste protótipo, precisa iniciar o Client da biblioteca OpenSSE e chamar a função setupDB com os parâmetros sendo: a localização dos logs utilizados, o número máximo de logs a serem percorridos e o caminho para o arquivo de saída. A chamada JSON-RPC retorna true caso tenha finalizado corretamente. A biblioteca OpenSSE possui a função de geração das chaves, para o escopo dos testes deste projeto foi gerado uma chave e utilizada essa mesma já gerada para os testes.

5.5. Paciente

O software usado pelo paciente possui uma chamada para geração da query pela chamada JSON-RPC GenerateQuery que recebe o arquivo JSON de preferências e retorna uma string com a query. A outra chamada JSON-RPC necessária pelo paciente é a da verificação dos resultados chamada CheckResult, que precisa como parâmetro as preferências do usuário no formato JSON e retorna uma string se houve ou não violação de acesso aos logs.

A geração dos logs cifrados do sistema médico descrita na seção 4.6 gera todos os logs, incluindo os que enviam dados (SUBMIT) e acessam dados (GET). Por isso, como

s-tag foi escolhido ser sempre a ACTION == GET já que a lógica entre as preferências de acesso do paciente serão trabalhadas em cima das x-tags e o JSON de preferências. Com essa s-tag o servidor operará somente nos indexes que acessam dados. Se for necessário realizar buscas com ACTION em um valor diferente de GET teria que ser feita uma outra query com a respectiva s-tag Para a geração das x-tags na query, foi criado a seguinte organização, para cada OB, adicionar os ACTORs que podem acessar em seguida. Sendo OB o objeto, como por exemplo X-RAY, e os ACTORs seguidos são os ACTORs que podem acessar esse OB.

No exemplo dado pelas preferências de John Doe a query ficaria da seguinte maneira:

OB:X-RAY ACTOR:1 ACTOR:2 ACTOR:3 ACTOR:4 OB:URINE-SAMPLE ACTOR:2 ACTOR:3 OB:DNA ACTOR:9 OB:MRI ACTOR:7 OB:LDL ACTOR:6 OB:HDL ACTOR:6 OB:VLDL ACTOR:6 OB:VHS ACTOR:1 OB:T3 ACTOR:1 ACTOR:4 OB:T4 ACTOR:1 ACTOR:4

Para isso, o método GenerateQuery precisa receber as preferências do usuário no formato Json. Abaixo está a função GenerateQuery. Para cada membro do array allow ele adiciona a string das x-tags a x-tag "OB: membro" e para cada ID permitido desse OB ele adiciona a x-tag ID: id, respeitando o formato OB, IDs permitidos

Com a string da query criada, é instanciado um objeto Client da biblioteca OpenSSE e chamada a função para criação da query conjunctiveSearch com base nessa string que retorna uma string com a query cifrada que pode ser enviada para o Server. Já que para o escopo do protótipo deste trabalho foram utilizadas uma chave previamente gerada, o Client já conhece o caminho dela. A outra chamada JSON-RPC necessária pelo Paciente é a de verificação de resultado. Com essa organização das x-tags essa função executa a mesma lógica para verificar as condições de acesso de cada x-tag. Para isso, ele precisa chamar o objeto do Client da biblioteca OpenSSE para decifrar o resultado. Como o caminho de onde está o resultado do verificador é conhecido nestes testes ele está diretamente no código. Essa função também necessita da chave previamente utilizada, como explicado anteriormente, ela já é conhecida nesse protótipo, então o caminho dela está diretamente no código.

O fetchResult retorna um vetor de dynamic bitsets que representam os resultados de cada x-tag. Esses bitsets são compostos por 0's e 1's que representam os índices que contém essa x-tag. Para verificar o resultado, o algoritmo percorre as preferências do usuário junto com o vetor de strings obtido pelo fetchResult, para cada resultado da x-tag do tipo ACTOR faz a operação booleana OR com um bitset temporário, depois é feito a operação booleana AND com o resultado do OB e então a negação desse resultado são os índices dos logs onde o acesso foi de um OB mas foi acessado por um ACTOR diferente do especificado na política de acesso, é feita a operação OR com um bitset de violações para o retorno. No final, ele verifica se houve algum índice que viola a política de acesso do usuário checando o bitset violations e retorna uma string com o resultado.

5.6. Verificador

O Verificador recebe o arquivo com os indexes e keywords cifrados com algoritmo de criptografia consultável e uma string com a query a ser realizada e retorna um arquivo com o resultado final. Para isso, o verificador precisa instanciar um objeto do Server

da biblioteca OpenSSE iniciar os indexes e keywords cifrados no objeto com o arquivo criado pelo Sistema Médico e então realizar a query enviada pelo cliente. Isso irá gerar um arquivo de resultado, como explicado na seção 4.2, que deve ser enviado para o cliente para gerar os resultados.

A execução da SearchQuery precisa da localização do arquivo com os indexes e keywords cifrados, a string com a query a ser realizada e a localização para o arquivo de saída. O SearchQuery inicializa uma instância do Server da OpenSSE e então faz a busca com a query recebida para o arquivo na localização especificada nos parâmetros.

5.7. Testes e Experimentos

Foram realizados testes com um número pequeno de logs para verificar a consistência dos resultados. Não foi levada em consideração a eficiência do algoritmo de criptografia consultável, por isso, foram utilizados somente 5 logs para cada teste. Ao executar os passos necessários pelas janelas do Sistema Médico, Paciente e Verificador a página do Verificador deve informar se houve ou não violação das preferências de acesso, e caso houver, mostrar uma lista dos índices de quais logs violaram as preferências.

No primeiro teste foram utilizados os seguintes logs de acesso:

ACTION:GET DATE:2018-02-14 ACTOR:1 OB:URINE-SAMPLE

ACTION:GET DATE:2018-02-14 ACTOR:2 OB:X-RAY

ACTION:GET DATE:2018-02-14 ACTOR:3 OB:X-RAY

ACTION:GET DATE:2018-02-14 ACTOR:4 OB:X-RAY

De acordo com as preferências definidas na seção 4.4, há violação no primeiro registro de acesso, onde o ACTOR:1 acessa as informações de URINE-SAMPLE, a qual só permite acessos dos ACTORS 2 e 3.

A página do Paciente gera a a query que deve ser enviada junto com o Encrypted Database, para o verificador que escreve um arquivo com os resultados. Estes resultados devem ser decifrados pela página do paciente a qual retornará a informação sobre a violação das preferências de acesso. Neste caso houve violação no log de índice 1 na lista dos logs. A resposta do software do paciente foi ”1 entry violates your privacy. 1000”

6. Conclusão

O uso de searchable encryption fora do contexto padrão da interação entre o cliente e a nuvem é muito interessante e promissor. Este trabalho proporcionou o desenvolvimento de uma biblioteca de criptografia consultável que torna possível explorar o uso de criptografia consultável em outras áreas.

Com a biblioteca OpenSSE, a implementação do sistema médico, cliente e principalmente a do verificador foram objetivas e claras. Com isso, seria simples outras entidades utilizarem o mesmo protocolo de criptografia consultável e comunicação definidos neste trabalho para implementar outros verificadores, aumentando a confiabilidade na verificação feitas pelos usuários de sistemas médicos.

Com o uso de um esquema de searchable encryption que não precise de uma verificação dos resultados pelo cliente teria um impacto maior pois poderiam ser feitas

arguições automáticas de dados disponíveis pelo sistema médico, como os dados cifrados de todos os clientes junto com a query que representa suas preferências.

A entidade que decida utilizar este esquema para prover uma maior confiabilidade de seu sistema tem que garantir a integridade da geração de logs de acesso assim como a sua integridade na geração dos logs cifrados. A confiabilidade do cliente nos verificadores pode ser melhorada com o número de implementações de verificadores e ele possa utilizar mais de um. O software que o cliente utiliza para criar a query e verificar o resultado também pode ser implementado em diferentes lugares além do que no próprio sistema médico. A confiabilidade no software que o cliente tem também pode ser melhorada por diferentes implementações disponíveis.

Este trabalho também poderia ser integrado a um framework de sistema médico para atingir diversas implementações consequentemente um número maior de usuários. Um frameworks de sistema médico disponível é o CareKit da Apple. Em 2017 a Apple lançou a possibilidade do compartilhamento de dados médicos na sua framework, porém depende da empresa implementando o sistema médico para definir políticas de privacidade e prover o usuário a capacidade de verificar essas políticas (DEKOSHKA, 2017). Com a integração deste trabalho com o framework desenvolvido pela Apple, seria possível transformar a implementação de qualquer sistema médico utilizando o CareKit seguro, fortalecendo privacidade dos usuários. Além disso, o CareKit é uma framework open-source e busca a colaboração de desenvolvedores como houve colaborações no desenvolvimento do compartilhamento de dados médicos (DEKOSHKA, 2017).

6.1. Trabalhos Futuros

Com o desenvolvimento deste trabalho, alguns pontos interessantes foram levados para serem desenvolvidos posteriormente. O primeiro deles, seria a implementação de outros algoritmos de criptografia consultável para a biblioteca OpenSSE. Com o surgimento de novos esquemas de criptografia consultável. O segundo, seria o estudo da aplicação de searchable encryption no contexto mais tradicional de banco de dados, a utilização da biblioteca OpenSSE seria muito interessante em uma implementação de um banco de dados SQL com suporte a searchable encryption para o cenário clássico dessa área entre cliente e nuvem. O terceiro, seria o estudo de novos esquemas de searchable encryption que não dependam de response-hiding para consultas booleanas expressivas. Dessa maneira seria possível eliminar a última interação com o cliente para validar o resultado neste trabalho.