

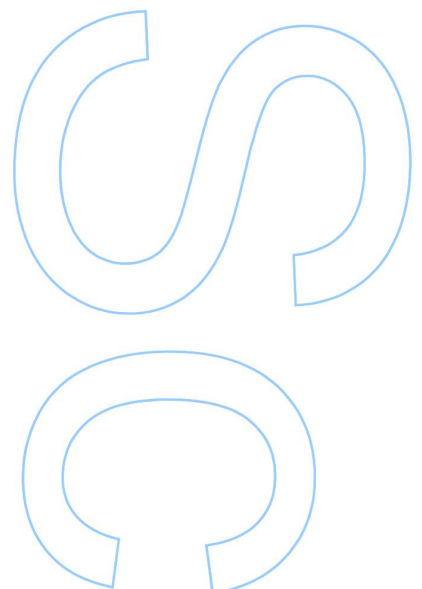
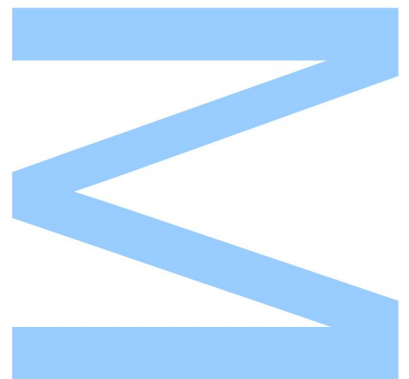
Base de dados segura para ambientes cloud

João Pedro Barros Mendes

Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2021

Orientador

Manuel Bernardo Martins Barbosa, Professor Associado, Universidade do Porto

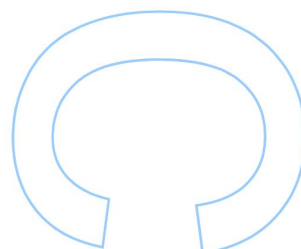
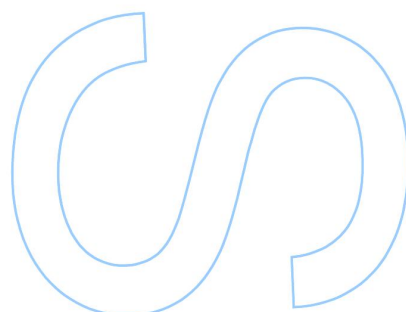
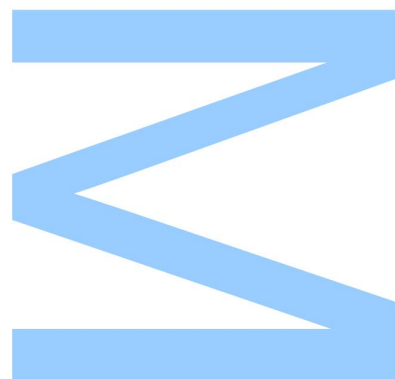




Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ____ / ____ / ____



Resumo

A LOQR fornece vários serviços aos clientes em que processa dados bancários por eles fornecidos, e posteriormente armazena-os em bases de dados na *cloud*. Esta empresa já coloca em prática medidas em relação à proteção desses dados, prevenindo contra um administrador malicioso dos serviços na *cloud* ou um atacante que consiga acesso à base de dados, sendo que parte dos dados que não serão usados para indexação serão cifrados pseudo-aleatoriamente.

Atualmente, nesse serviço, a segurança dos dados fornecidos depende de uma chave mestra. Neste trabalho pretende-se especificar e desenvolver um sistema que possa ser integrado no sistema atual da LOQR, e que cumpra um requisito fundamental: que o controlo da chave de cifra de dados do cliente seja passada para o mesmo, de modo a que este em qualquer instante consiga negar o acesso a dados em plaintext.

Numa primeira fase vão ser descritos sistemas que funcionam de forma semelhante, problemas com a cifragem de dados em bases de dados, a arquitetura do sistema e todos os seus componentes. Em função dessas informações obtidas, será possível propor e analisar soluções que visam cumprir o principal requisito pretendido, e justificar qual a solução mais adequada. Também é desenvolvido um protótipo que simule essa solução, e que demonstre o controlo que o cliente passa a ter com a chave na sua posse.

Agradecimentos

Ao Prof. Manuel Barbosa pelo apoio ao longo do desenvolvimento desta dissertação, e pela capacidade de propor metas interessantes ao longo deste trabalho. Também agradeço à LOQR, em especial ao Helder Sousa, João Cerdeira, Nuno Santos e Pedro Borges, pelo interessante tema proposto, por toda a disponibilidade e clareza ao longo das nossas reuniões.

Adicionalmente, agradeço aos meus pais Maria do Carmo e Valdemar Mendes, ao meu irmão Pedro Mendes e aos meus amigos próximos pelo apoio e acompanhamento ao longo deste trabalho.

Conteúdo

Resumo	i
Agradecimentos	iii
Conteúdo	vii
Lista de Tabelas	ix
Lista de Figuras	xiii
Lista de Blocos de Código	xv
Acrónimos	xvii
1 Introdução	1
1.1 Descrição Geral	1
1.2 Objetivos	2
1.3 Estrutura do Documento	2
2 Enquadramento Tecnológico	5
2.1 Primitivas Criptográficas	5
2.1.1 Funções de Hash	5
2.1.2 Message Authentication Code (MAC)	9
2.1.3 Cifras de Bloco	10
2.1.4 Modos de Operação	11

2.1.5	Primitivas Assimétricas	11
2.2	Indexação, Serviços na Cloud e Tecnologias Necessárias para a Implementação	14
3	Trabalho Relacionado	19
3.1	Modelização de Ataques	19
3.1.1	Cifras em Sistemas Legacy	21
3.1.2	Ataques a Dados Cifrados	21
3.2	Computação Segura na Base de Dados	22
3.3	Pré-Processamento de Dados	28
4	Análise de Requisitos	33
4.1	Tecnologias Integradas	33
4.1.1	Always Encrypted	33
4.1.2	S3	36
4.2	Especificação da Arquitetura Atual	36
4.2.1	Agentes	36
4.2.2	Arquitetura SQL Server	37
4.2.3	Arquitetura AWS	40
4.2.4	Ativos	43
4.2.5	Modelo de Confiança	44
4.3	Requisitos da Solução	45
5	Concepção	47
5.1	Racional	47
5.2	Solução Proposta	48
5.2.1	Decisões Tecnológicas	49
5.2.2	Arquitetura	51
5.2.3	Fluxos de Dados e Ações	53
5.3	Cumprimento dos Requisitos	56

6	Implementação e Resultados Experimentais	59
6.1	Primeira Fase - Estado Atual	59
6.1.1	Arquitetura	59
6.1.2	Fluxos de Dados e Ações	67
6.2	Segunda Fase - Chaves no Key Vault e no KMS	71
6.2.1	Arquitetura, Fluxos de Dados e Ações	71
6.2.2	KMS	71
6.2.3	Key Vault	79
7	Conclusão	87
7.1	Trabalho Futuro	88
	Bibliografia	89

Lista de Tabelas

5.1	Cumprimento dos requisitos do Sistema Global	57
5.2	Cumprimento dos requisitos do Serviço de Gestão de Chaves Criptográficas . . .	58

Lista de Figuras

2.1	Estrutura base do Secure Hash Algorithm.	6
2.2	Esquema Hash-based Message Authentication Code.	9
2.3	Modo de operação CBC.	12
2.4	Modo de operação GCM.	12
2.5	Estrutura do <i>Optimal Asymmetric Encryption Padding</i>	15
2.6	Exemplo de conexão e <i>query</i> no Azure Data Studio.	16
3.1	Arquitetura e sequência de operações da TrustedDB	24
3.2	Arquitetura da CipherBase	25
3.3	Arquitetura da EnclaveDB	27
3.4	Arquitetura da CryptDB	28
3.5	Onions que permitem vários níveis de encriptação	31
3.6	Cliente faz um pedido em que é necessário remover uma layer	31
4.1	Arquitetura do serviço de armazenamento que integra o SQL Server.	38
4.2	Fluxo de dados do serviço de armazenamento que integra o SQL Server.	39
4.3	Arquitetura do serviço de armazenamento que integra o S3.	41
4.4	Fluxo de dados do serviço de armazenamento que integra o armazenamento S3.	42
4.5	Arquitetura Base para cumprir requisitos.	45
5.1	UML esquema da base de dados.	50
5.2	Arquitetura do Sistema Proposto.	51

5.3	Ações e Fluxos de Dados do Sistema Proposto.	54
6.1	Arquitetura da Primeira Fase do Protótipo.	60
6.2	Menu do Operador.	62
6.3	Página de <i>login</i>	62
6.4	Página de <i>logout</i>	62
6.5	Página para adicionar contas novas.	62
6.6	Página para listar contas associadas.	63
6.7	Página onde se pode fazer <i>upload</i> de ficheiros.	63
6.8	Página onde se pode fazer <i>download</i> de ficheiros.	63
6.9	Script para gerar ficheiros P12.	65
6.10	Exemplo de acesso à keystore através da aplicação.	65
6.11	Cifra de um ficheiro fornecido pelo cliente.	66
6.12	Decifra de um ficheiro que o cliente pretende fazer <i>download</i>	66
6.13	Configuração básica do SQL Server na Azure.	67
6.14	Configuração básica do SQL Server na AWS.	67
6.15	Comandos SQL para criar as tabelas nas bases de dados.	68
6.16	Dados em repouso na base de dados.	68
6.17	Ações e Fluxos de Dados da Primeira Fase do Protótipo.	69
6.18	Ficheiro de credenciais de acesso aos serviços.	72
6.19	Criação de uma chave no KMS.	72
6.20	Dar permissões de uso a um utilizador no KMS.	73
6.21	Cifra de uma chave AES na segunda fase do protótipo, utilizando pedidos feitos ao KMS.	74
6.22	Decifra de uma chave AES na segunda fase do protótipo, utilizando pedidos feitos ao KMS.	74
6.23	Como fazer <i>disable</i> a uma chave através da consola do KMS.	75
6.24	<i>Print</i> da excepção na Aplicação Web.	75
6.25	Resultado que o Cliente visualiza depois de a chave ser <i>disable</i>	75

6.26 Criação de um <i>dashboard</i> para ver acessos a chaves no KMS.	76
6.27 Configurar um alarme no KMS.	76
6.28 Definir a métrica que deve ser observada num alarme no KMS.	77
6.29 Definir a condição para um alarme ser ativado no KMS.	77
6.30 Selecionar o tipo de notificação associado a um alarme no KMS.	78
6.31 Exemplo de um e-mail recebido após um alarme ser ativado no KMS.	78
6.32 Exemplo de um e-mail com logs de atividades ocorridas depois de um alarme ter sido ativado no KMS.	78
6.33 Criação de uma chave num Key Vault.	79
6.34 Definir permissões de acesso a uma Key Vault.	80
6.35 Registo e atribuição de permissões através do Cloud Shell.	80
6.36 Cifra de uma chave AES na segunda fase do protótipo, utilizando pedidos feitos ao Key Vault.	81
6.37 Decifra de uma chave AES na segunda fase do protótipo, utilizando pedidos feitos ao Key Vault.	81
6.38 Como fazer <i>disable</i> a uma chave através do portal da Azure.	82
6.39 <i>Print</i> da exceção na Aplicação Web.	83
6.40 Monitorizar métricas de um Key Vault.	83
6.41 Criar um grupo para ser associado a um alarme de um Key Vault.	84
6.42 Adicionar um mail a um grupo de um alarme de um Key Vault.	84
6.43 Definir a condição para um alarme de um Key Vault ser ativado.	84
6.44 Atribuir definições adicionais a uma alarme de um Key Vault.	84
6.45 Exemplo de um e-mail de quando um alarme é ativado.	85

Lista de Blocos de Código

Acrónimos

3DES	Triple Data Encryption Algorithm	ECDSA	Elliptic Curve Digital Signature Algorithm
AEAD	Authenticated Encryption with Associated Data	EM	Encoded Message
AES	Advanced Encryption Standard	EPC	Enclave Page Cache
API	Application Programming Interface	FIPS	Federal Information Processing Standards
AWS	Amazon Web Services	FPGA	Field-programmable Gate Array
BLOB	Binary Large Object	GCM	Galois/Counter Mode
CBC	Cipher Block Chaining	GF	Galois Field
CCA	Chosen Ciphertext Attack	HMAC	Hash-based Message Authentication Code
CEK	Column Encryption Key	HOM	Homomorphic Encryption
CMK	Column Master Key	HSM	Hardware Security Module
CN	Common Name	HTTPS	Hyper Text Transfer Protocol Secure
CPU	Central Processing Unit	HTTP	Hyper Text Transfer Protocol
CS	Componente Seguro	IAM	Identity and Access Management
DBMS	Database Management System	IP	Internet Protocol
DB	Data Block	ISP	Internet service provider
DEK	Data Encryption Key	IV	Initialization Vector
DES	Data Encryption Standard	KMS	Key Management Service
DET	Deterministic Encryption	MAC	Message Authentication Code
ECB	Electronic Codebook	MGF	Mask Generating Function
ECC	Elliptic-curve cryptography	NIST	National Institute of Standards and Technology

OAEP	Optimal Asymmetric Encryption Padding	SDK	Software Development Kit
ODBC	Open Database Connectivity	SGX	Software Guard Extensions
OPE	Order-preserving Encryption	SHA	Secure Hash Algorithm
OS	Operating System	SQL	Structured Query Language
PCI	Peripheral Component Interconnect	SaaS	Software as a service
PKCS	Public Key Cryptography Standards	TLS	Transport Layer Security
PRP	Pseudorandom Permutation	TM	Trusted Module
PaaS	Platform as a Service	UDF	User-defined Functions
RAM	Random Access Memory	UML	Unified Modeling Language
RND	Random	UM	Untrusted Machine
RSA	Rivest–Shamir–Adleman	VM	Virtual Machine
S3	Simple Storage Service	XML	Extensible Markup Language
SCPU	Secure Coprocessors	XOR	Exclusive OR

Capítulo 1

Introdução

Nestas últimas décadas com o desenvolvimento de todos os tipos de serviços informáticos surge também a necessidade de armazenar dados em dimensões bastante grandes. Como tal, empresas que não se dedicam principalmente a áreas tecnológicas, ou mesmo as que se dedicam, recorrem a *outsourcing* para possuírem esses tipos de serviços, mas sem terem de ser elas a implementá-los e a geri-los. Este processo acaba também por reduzir os custos de alguns serviços dado que o investimento inicial já foi feito por uma outra empresa especializada nessa área.

Com o desenvolvimento informático surgem também, cada vez mais, vetores de ataques a estas bases de dados. Cifrar tudo o que é possível não se apresenta como uma solução realista e funcional, pois existem mecanismos presentes nestas bases de dados que permitem uma pesquisa otimizada de dados que podem não ser aplicáveis, dependendo da cifra usada. Contudo, certas cifras podem ser aplicadas que acabam por preservar algumas propriedades em relação aos dados antes de serem cifrados. No entanto, essas cifras acabam por revelar alguma informação sobre os dados em *plaintext*.

Mas mesmo que apenas alguns dados sejam cifrados, e que a forma como são cifrados seja totalmente aleatória, existe sempre um problema em questão: a manutenção da chave é essencial para a segurança desses dados. Surgem então construções onde as chaves são cifradas depois de serem usadas, e guardadas em conjunto com os dados cifrados por essas mesmas chaves, mas aí surge mais uma vez a questão da gestão dessa *master key*.

1.1 Descrição Geral

A LOQR, atualmente, oferece serviços de armazenamento de dados aos seus clientes, onde resumidamente, utiliza bases de dados hospedadas na Cloud e várias instâncias de servidores que funcionam como intermediários entre os clientes e as bases de dados. Todo o processo de conexão e pesquisas feitas às bases de dados são feitas a partir desses servidores intermédios, em resposta a pedidos feitos pelos clientes. Parte desses dados são confidenciais, e como tal, não devem ser armazenados em serviços na *cloud* em *plaintext*, e, portanto, são cifrados do lado do

servidor, através de uma chave gerida por operadores da LOQR. Portanto, o cliente acaba por não ter qualquer interação com o processo de cifragem de dados, nem da chave em questão.

1.2 Objetivos

Como foi referido na secção anterior, a responsabilidade de criação, manutenção e eliminação da chave cai sobre os operadores da LOQR. O principal objetivo nesta dissertação é passar o controlo dessa chave para o cliente, mas de uma forma em que a manutenção dessa chave seja fácil e que o processo de cifra seja feito de forma transparente para os clientes que estão a usar a aplicação que interage com o sistema. É então também priorizada a confidencialidade dos dados, e que após o momento em que o cliente indisponibiliza o acesso à chave, o sistema não consiga aceder a dados assinalados como confidenciais. É de notar que apesar de algumas das construções criptográficas utilizadas no sistema atual garantirem integridade dos dados, e de a chave usada para integridade e confidencialidade ser a mesma, é do interesse da LOQR que o foco deste projeto seja apenas na confidencialidade dos dados. De forma idêntica, a disponibilidade dos dados não será analisada.

O objetivo inicial é identificar qual é o estado da arquitetura atual do serviço da LOQR, quais os atores que interagem com esse serviço, como os dados fluem nessa arquitetura, quais os processos de cifra e todas as chaves criptográficas envolvidas, e quais os requisitos que terão de ser cumpridos para alterar para o sistema referido no parágrafo anterior.

O objetivo final será propor uma solução teórica que alcance os requisitos pretendidos, e posteriormente, desenvolver um protótipo que numa primeira fase represente o estado atual do serviço oferecido pela LOQR, tentando representar de uma forma realista os componentes atuais, e que numa segunda fase represente essa solução, alterando assim a parte de gestão de chaves criptográficas. Neste protótipo serão representados os pontos críticos desta infraestrutura e vai ser possível analisar dados em repouso, para demonstrar a confidencialidade alcançada nesses mesmos pontos. Também será demonstrada a capacidade que o cliente tem de vedar o acesso à chave, e assim indisponibilizar dados confidenciais.

1.3 Estrutura do Documento

Nesta secção será descrita a estrutura deste documento, referindo os conteúdos de cada capítulo:

- **Capítulo 2 - Conhecimentos Necessários:** Aqui serão descritos, resumidamente, conceitos que deverão ser do conhecimento do leitor, para que consiga entender os próximos capítulos. Um leitor com um elevado conhecimento em Segurança Informática poderá passar dispensar a leitura deste capítulo.
- **Capítulo 3 - Trabalho Relacionado:** Existem várias aproximações que têm sido propostas

para resolver problemas relativamente ao armazenamento seguro em bases de dados. Aqui serão apresentadas diferentes aproximações e modelos de ataque a base de dados.

- Capítulo 4 - Análise de Requisitos: Neste capítulo será analisado o estado atual do serviço oferecido pela LOQR e as tecnologias usadas neste serviço. Será então possível especificar os requisitos pretendidos para o futuro sistema.
- Capítulo 5 - Concepção: Inicialmente serão expostas quais as possíveis aproximações que se podem considerar para alcançar soluções em que o cliente tenha o controlo da chave, e relacioná-las com os requisitos. Depois será proposta uma solução que vá ao encontro dos requisitos, e finaliza-se com uma justificação de que essa solução cumpre os requisitos.
- Capítulo 6 - Implementação e Resultados Experimentais: Aqui será descrito o desenvolvimento de um protótipo da solução anterior, descrevendo alguns pormenores técnicos e as suas principais funcionalidades, nomeadamente, de como o cliente consegue negar o acesso à sua chave mestra, e quais os resultados de tal ação.
- Capítulo 7 - Conclusão: É feito um resumo do que se conseguiu alcançar nesta dissertação e como o sistema poderá continuar a evoluir.

Capítulo 2

Enquadramento Tecnológico

Para entender conceitos que vão ser debatidos ao longo desta dissertação, é necessário que o leitor tenha conhecimentos na área da Informática e da Segurança Informática. Neste capítulo serão apresentados, a um alto nível, certos conceitos que deverão ser entendidos para que o leitor consiga acompanhar os próximos capítulos. Um leitor bastante entendido nestas áreas, poderá passar diretamente para o próximo capítulo.

2.1 Primitivas Criptográficas

Nas próximas subsecções serão descritas diferentes primitivas que visam preservar a confidencialidade, integridade e autenticidade de dados informáticos, que são usadas em sistemas da LOQR e que serão utilizadas na implementação da solução proposta.

2.1.1 Funções de Hash

As Funções de *hash* [1] são funções que permitem mapear uma mensagem de tamanho ilimitado, para um valor de tamanho limitado.

Tais funções têm várias funcionalidades, mas no contexto desta dissertação apenas são abordadas por questões de integridade e autenticidade. Em específico serão descritas as propriedades do conjunto das funções *hash* criptográficas, em que é possível perceber se *bits* na mensagem original foram alterados, dado que a alteração de um *bit* na mensagem propaga-se para o valor da *hash*. Como tal, para uma função H ser considerada uma função *hash* criptográfica, deverá seguir as seguintes propriedades:

- Tamanho de *input* variável.
- Tamanho de *output* fixo.
- Fácil de computar para um determinado $H(x)$, qualquer que seja x .

- Resistência à pré-imagem: para uma determinada hash h , deverá ser computacionalmente inviável encontrar y , tal que $H(y)=h$.
- Resistência à segunda pré-imagem: para uma mensagem x , deverá ser computacionalmente inviável encontrar $y \neq x$, tal que $H(y)=H(x)$.
- Resistência a colisões: deverá ser computacionalmente inviável encontrar y e x ($y \neq x$), tal que $H(y)=H(x)$.

2.1.1.1 SHA-256

Uma das funções mais integradas nos sistemas atuais é o SHA-256 [2], pertencente à família de funções SHA-2 (*Secure Hash Algorithm 2*). Esta função segue as propriedades descritas em cima, e possui um *message digest* (resultado final) de 256 *bits* e um limite de 2^{64} *bits* (2 091 752 *terabytes*). É assim possível gerar uma *hash* com uma probabilidade baixa de colisões, não reversível e com um *output* pseudo-aleatório. O esquema base das funções SHA pode ser visto na Figura 2.1.

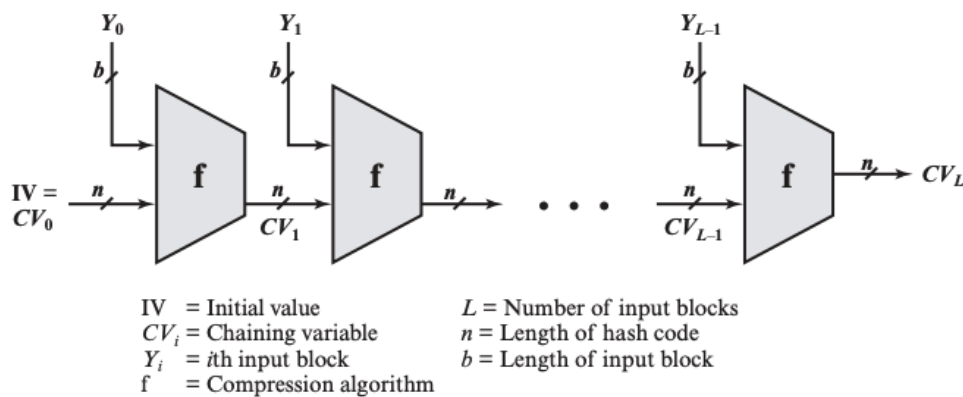


Figura 2.1: Estrutura base do Secure Hash Algorithm. [1]

Para gerar a *hash* de uma determinada mensagem M serão necessários os seguintes passos:

1. Para que seja possível dividir M , é adicionado *padding* de modo a que a mensagem seja um múltiplo de 512 da seguinte forma:

Considerando l como o tamanho de M (em *bits*), concatena-se de um "1" à mensagem, seguido também de k "0" *bits*, em que k é a solução para a equação $l + 1 + k \equiv 448 \pmod{512}$. Adicionalmente são concatenados os restantes 64 *bits* (para alcançar os 512) que representam l em binário.

2. A Mensagem é então dividida em N blocos de 512 *bits*: $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$. Como um bloco de 512 *bits* pode ser dividido em 16 blocos de 32 *bits*, os primeiros 16 *bits* de um bloco i são representados por $M_0^{(i)}$, os seguintes 16 por $M_1^{(i)}$, e assim continuamente.

3. O valor inicial da *hash* H^0 é atribuído. Para garantir que estes valores não são escolhidos de uma forma que afirme ser aleatória, mas que depois possa ser usada por um atacante para explorar a função, foram escolhidos como valores os primeiros 32 *bits* da parte fracionária das raízes quadradas dos primeiros 8 números primos (2, 3, 5, 7, 11, 13, 17, 19). Os valores iniciais são os seguintes (em hexadecimal):

$$\begin{aligned} H_0^{(0)} &= 6a09e667 \\ H_1^{(0)} &= bb67ae85 \\ H_2^{(0)} &= 3c6ef372 \\ H_3^{(0)} &= a54ff53a \\ H_4^{(0)} &= 510e527f \\ H_5^{(0)} &= 9b05688c \\ H_6^{(0)} &= 1f83d9ab \\ H_7^{(0)} &= 5be0cd19. \end{aligned}$$

Também, são criadas 64 constantes de 32 *bits* $K_0^{256}, K_1^{256}, \dots, K_{63}^{256}$. Estas também têm um processo de escolha de forma semelhante aos valores iniciais da *hash*, dado que são os primeiros 32 *bits* da parte fracionária das raízes cúbicas dos primeiros 64 números primos.

4. Para prosseguir com a computação da *hash*, é necessário definir funções que recebem como *input words* de 32 *bits*, e que produzam um *output* também de 32 *bits*. Para tal são utilizadas as seguintes funções lógicas (em que *ROTR* representa um *rotate right* e *SHR* um *shift right*):

- $Ch(x, y, z) = (x \wedge y) \oplus (x \wedge z)$
- $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
- $\sum_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
- $\sum_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$
- $\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$
- $\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

, que possibilitam computar a *hash* resultante. Portanto, para todas as mensagens M^i , em que i vai de 1 até N , são executados os seguintes passos (onde a adição é feita módulo 2^{32}):

- 4.1. Criar o planeamento da mensagem adicionando 48 novas *words* da seguinte forma:

$$W_t = \begin{cases} M_t^{(i)}, & 0 \leq t \leq 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16}, & 16 \leq t \leq 63 \end{cases}$$

- 4.2. Inicializar oito variáveis **a**, **b**, **c**, **d**, **e**, **f**, **g** e **h** com os valores anteriores da *hash* (H^{i-1}):

- $a = H_0^{(i-1)}$

- $b = H_1^{(i-1)}$
- $c = H_2^{(i-1)}$
- $d = H_3^{(i-1)}$
- $e = H_4^{(i-1)}$
- $f = H_5^{(i-1)}$
- $g = H_6^{(i-1)}$
- $h = H_7^{(i-1)}$

4.3. Usar cada *word* das 64 do planeamento da mensagem, de $t=0$ até $t=63$, para alterar os valores das variáveis anteriores (recorrendo a duas variáveis temporárias $T1$ e $T2$):

- $T1 = h + \sum_1^{256}(e) + Che(e, f, g) + K_t^{256} + W_t$
- $T2 = \sum_0^{256}(a) + Maj(a, b, c)$
- $h = g$
- $g = f$
- $f = e$
- $e = d + T1$
- $d = c$
- $c = b$
- $b = a$
- $a = T1 + T2$

5. Computar a respetiva *hash* intermédia $H^{(i)}$ computada através do bloco $M^{(i)}$, constantes, e a *hash* anterior:

- $H_0^{(i)} = a + H_0^{(i-1)}$
- $H_1^{(i)} = b + H_1^{(i-1)}$
- $H_2^{(i)} = c + H_2^{(i-1)}$
- $H_3^{(i)} = d + H_3^{(i-1)}$
- $H_4^{(i)} = e + H_4^{(i-1)}$
- $H_5^{(i)} = f + H_5^{(i-1)}$
- $H_6^{(i)} = g + H_6^{(i-1)}$
- $H_7^{(i)} = h + H_7^{(i-1)}$

Quando se processar o último bloco M^N , obtém-se a *message digest* de 256 *bits* da mensagem que será a concatenação de partes da *hash* intermédia do último bloco:

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)} \parallel H_7^{(N)}$$

2.1.2 Message Authentication Code (MAC)

Como foi visto na secção anterior, é demonstrado como se pode ver se uma mensagem está íntegra. No entanto, isto não impede um atacante de reformular uma mensagem e recalculer a sua *hash*. Como tal, são necessários mecanismos que, para além de integridade, ofereçam autenticidade.

Como as funções de *hash* não foram desenvolvidas para que possam receber uma chave como *input*, surgem esquemas HMAC (*Hash-based Message Authentication Code*) [1], onde se recorre a uma função de *hash* (p.e. SHA-256), mas fornecendo-lhe como *input* também uma chave. É assim possível alcançar os seguintes objetivos:

- Usar funções de *hash* já implementadas e que têm reconhecimento.
- Alterar a função de *hash* facilmente no caso de surgirem funções que se apresentem como melhores, quer em termos de velocidade, quer de segurança.
- Preservar as capacidades já demonstradas da função de *hash* usada.
- Recorrer à segurança da função como âncora de segurança deste esquema.

Para gerar uma *tag* de uma mensagem recorre-se aos passos que podem ser vistos na Figura 2.2, com a respetiva legenda:

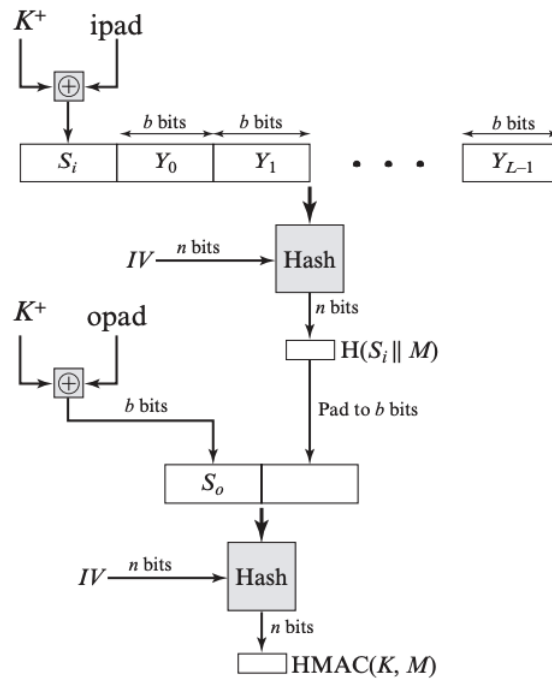


Figura 2.2: Hash-based Message Authentication Code. [1]

H - Função de *hash* usada (por exemplo, SHA-256);

IV - Vetor de inicialização para a função de *hash*;

M - Mensagem de *input*;

Y_i - Bloco número i de M ;

L - Número de blocos em M ;

b - Número de *bits* num bloco;

n - Tamanho da *hash* gerada por H ;

K - Chave secreta. Se a chave for maior do que b deverá ser usada como *input* para uma função de *hash* para gerar uma chave menor;

K^+ - K com zeros adicionados à esquerda para que gere um bloco de b *bits*;

ipad - "00110110"repetido $b/8$ vezes;

opad - "01011100"repetido $b/8$ vezes.

Como tal, a função resultante deste esquema pode-ser descrita por:

$$\text{HMAC}(K, M) = H[(K^+ \oplus \text{opad}) \parallel H[(K^+ \oplus \text{ipad}) \parallel M]],$$

e pode ser descrita através dos seguintes passos:

1. Adicionar zeros à esquerda de K para gerar K^+ , com tamanho b .
2. XOR (*Exclusive OR*) entre K^+ e *ipad*, resultando S_i .
3. Concatenar M com S_i .
4. Fornecer como *input* à função H a concatenação de M com S_i .
5. XOR entre K^+ e *opad*, resultando S_0 .
6. Concatenar o resultado do passo 4 com S_0 .
7. Calcular a *hash* do resultado do passo anterior.

Por vezes este esquema é usado com o esquema AEAD (*Authenticated Encryption with Associated Data*), em que o MAC de uma determinada mensagem é processado ao mesmo tempo, antes, ou depois do processo de encriptação da mensagem. Um utilizador poderá assim, para além de decifrar a mensagem, verificar que quem a criou tem acesso à chave, pois caso contrário, seria impossível gerar o MAC.

2.1.3 Cifras de Bloco

O AES [3] (*Advanced Encryption Standard*) é um *standard* estabelecido pelo NIST (*National Institute of Standards and Technology*) para proteção de dados informáticos, que especifica o

algoritmo Rijndael. Com este algoritmo é possível cifrar e decifrar dados através da mesma chave, uma chave simétrica. O resultado da cifra é referido como *ciphertext*, enquanto que os dados originais e o resultado da decifra são denominados de *plaintext*.

São computados blocos de 128 *bits* [1], usando chaves de 128, 192 ou 256 *bits*, em que o tamanho da chave irá influenciar o número de rondas, podendo ser de 10,12 ou 14, respetivamente.

Inicialmente, a chave fornecida é expandida para que seja gerada uma sub-chave para cada ronda. Posteriormente, em cada ronda, é feita uma substituição de *bytes* de acordo com a S-box (uma tabela pré-definida no *standard*), uma permutação de linhas do *estado* (estado intermédio do processo), uma substituição de colunas recorrendo a aritmética em $GF(2^8)$, e por fim, faz-se XOR do *estado* com a respetiva sub-chave de ronda. O resultado final será o *ciphertext* desse bloco. Deste modo, os primeiros três processos fornecem confusão, difusão e não linearidade, e o processo de XOR com a chave adiciona confidencialidade.

2.1.4 Modos de Operação

No entanto, dessa forma apenas é possível cifrar um bloco, e portanto, acabam por ser aplicados modos de operação para que seja possível cifrar vários blocos. Alguns modos requerem que exista um vetor de inicialização (IV) que funciona como um "bloco inicial", e também existem modos que necessitam de *padding* caso os *bytes* do *plaintext* não completem os 128 *bits* de um bloco.

Existem vários modos de operação [4] [5], mas nesta dissertação apenas são referidos os seguintes:

- CBC (*Cipher block chaining*; Figura 2.3) Existe a operação XOR entre um bloco de *plaintext* e o bloco anterior cifrado, e portanto, este modo necessita de um IV.
- GCM (*Galois/counter mode*; Figura 2.4) Este modo é um *counter mode* que contém um processo de autenticação. Aqui o IV é usado para criar um *counter* que é incrementado de modo a obter blocos diferentes.

2.1.5 Primitivas Assimétricas

Em sub-seções anteriores foram apresentadas formas de preservar confidencialidade para dados usando apenas uma chave. No entanto, em certas circunstâncias é conveniente que uma parte transmita a uma outra parte uma mensagem, garantindo que apenas essa consiga decifrar. Portanto, nesta sub-seção será apresentada uma função que permite que tal aconteça.

2.1.5.1 Função RSA

O RSA [7](criado por Rivest, Shamir e Adleman) é um algoritmo em que existe um par de chaves assimétrico, composto por uma chave pública e uma chave privada.

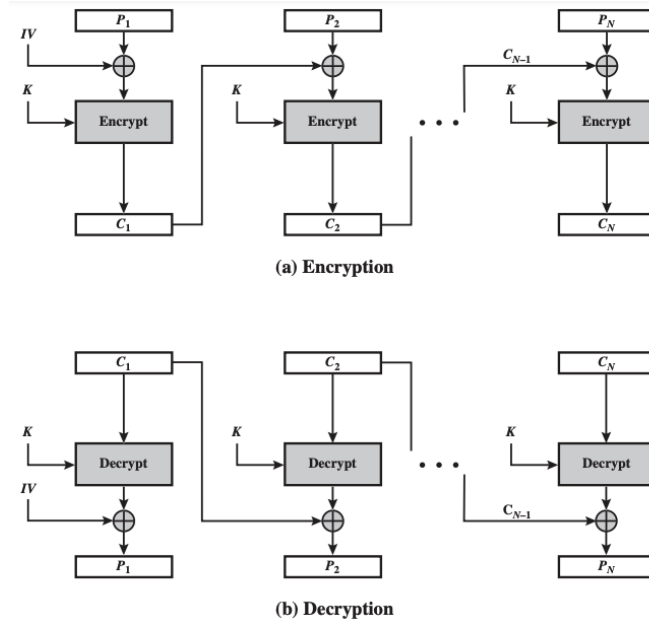


Figura 2.3: Modo de operação CBC [1].

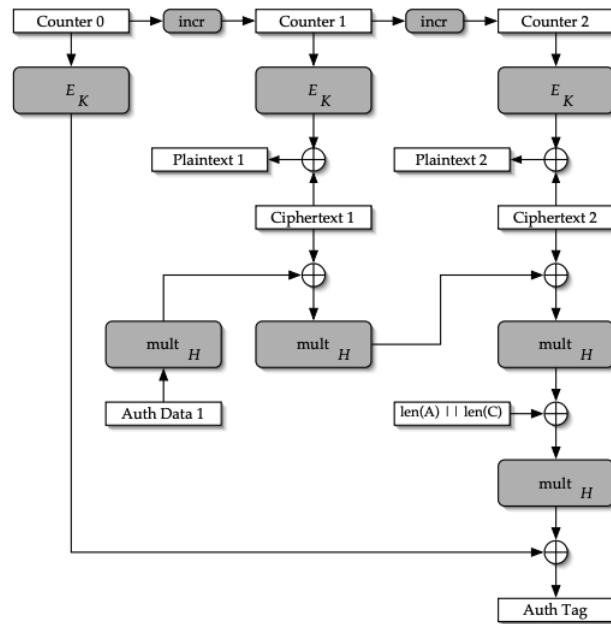


Figura 2.4: Modo de operação GCM [6].

A chave privada é composta por um valor d que apenas o dono da chave conhece, e a chave pública por um valor e . Existe ainda um n que ambas as partes conhecem [1]. A cifração e decifração representam-se da seguinte forma, em que M representa a mensagem em *plaintext* e C o *ciphertext* resultante:

$$C = M^e \bmod n$$

$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

Para que tal seja possível de acontecer, é necessário que sejam escolhidos dois números coprimos p e q , em que $n = p \times q$, e que e e d sejam inversos multiplicativos em $\varphi(n)$ (Função de Euler) e que sejam coprimos de $\varphi(n)$, dado que :

$$ed \bmod \varphi(n) = 1, \text{ o que significa o mesmo que}$$

$$ed \bmod \varphi(n) \equiv 1$$

$$d \equiv e^{-1} \bmod \varphi(n) ,$$

e também que

$$\varphi(pq) = (p - 1)(q - 1)$$

Portanto, a chave pública será o par $\{n, e\}$ e a chave privada $\{n, d\}$. Conhecendo p , q e consequentemente $\varphi(n)$, é possível descobrir d , e como tal, apesar de estes componentes não representarem a chave privada, deverão ser mantidos secretos.

Este algoritmo é usado para várias funcionalidades, no entanto tem algumas limitações, sendo a velocidade bastante inferior à do AES, e é também limitado no tamanho dos dados cifrados.

Como tal, uma das funcionalidades deste sistema é poder usar uma chave pública para cifrar uma chave AES que tenha sido usada para proteger dados, e depois decifrar a chave AES com a chave privada. Isto tem a vantagem de qualquer pessoa conseguir cifrar essa chave AES, mas apenas o dono da chave privada a conseguirá decifrar depois. Desta forma também é possível armazenar chaves AES cifradas em locais de insegurança.

Esta função e o esquema apresentado na próxima sub-secção são especificados no *standard* PKCS #1 [8] (*Public Key Cryptography Standards*), onde são detalhadas recomendações para a implementação de criptografia de chave pública, nomeadamente, primitivas criptográficas, esquemas de encriptação e de assinaturas, baseados na função RSA.

2.1.5.2 CCA e RSA-OAEP

Apesar da utilidade da função apresentada na sub-secção anterior, esta é vulnerável a um ataque *Chosen Ciphertext Attack*. Este ataque pressupõe que o atacante pode fazer pedidos de *ciphertexts* à sua vítima, e que esta lhe devolve o *plaintext* correspondente. Dessa forma, e criando *ciphertexts* específicos, é possível determinar o *plaintext* de blocos cifrados anteriormente, e possivelmente, cifrados por uma outra parte.

Para o atacante conseguir decifrar um determinado $C = M^e \bmod n$ deverá :

1. Computar $X = (C \times 2^e) \bmod n$.

2. Fornecer X à vítima e receber $Y = X^d \bmod n$, sendo que:

$$X = (C \bmod n) \times (2^e \bmod n) = (M^e \bmod n) \times (2^e \bmod n) = (2M)^e \bmod n.$$

3. O atacante recebe $Y = (2M) \bmod n$ e consegue assim recuperar M .

Para dificultar este ataque surge o OAEP (*Optimal Asymmetric Encryption Padding*), um esquema que não permite que um atacante consiga manipular a função e assim obter um *plaintext* à sua escolha.

Uma mensagem pode ser transformada através dos seguintes passos (Figura 2.5):

1. Um parâmetro P que é usado como *input* para uma função de $hash(H)$. Este parâmetro é opcional, e se não for fornecido, será uma *string* vazia [9].
2. O resultado é concatenado com zeros e com um octeto com o valor "0x01". Este *padding* tem tamanho variável, de acordo com o tamanho que se queira para o bloco final.
3. A mensagem (M) é concatenada ao produzido anteriormente e é assim gerado o *data block DB*: $DB = H(P) || Padding || M$.
4. Uma *seed* aleatória é gerada e é fornecida como *input* para uma função *hash* chamada *mask generating function (MGF)*.
5. O resultado desta última *hash* e o DB são usados para calcular um *XOR*, resultando o *maskedDB*.
6. O *maskedDB* é usado como *input* para uma *MGF*, sendo que a *hash* resultante será usada juntamente com a *seed* para uma operação *XOR*. O resultado deste *XOR* é o *maskedSeed*.
7. A *maskedSeed* e o *maskedDB* são concatenados para gerar a *encoded message(EM)*. Esta mensagem será a que é fornecida à função RSA.

É de notar que de um EM é possível recuperar a mensagem original através dos seguintes passos ($\oplus$ representa um XOR):

1. $seed = maskedSeed \oplus MGF(maskedDB)$
2. $DB = MGF(seed) \oplus maskedDB$, de onde se retirar o *padding* e se pode recuperar M .

2.2 Indexação, Serviços na Cloud e Tecnologias Necessárias para a Implementação

As bases de dados criam estruturas adicionais para permitirem que pesquisas sejam otimizadas, de modo que, quando uma pesquisa é feita, uma tabela não tenha de ser percorrida na sua

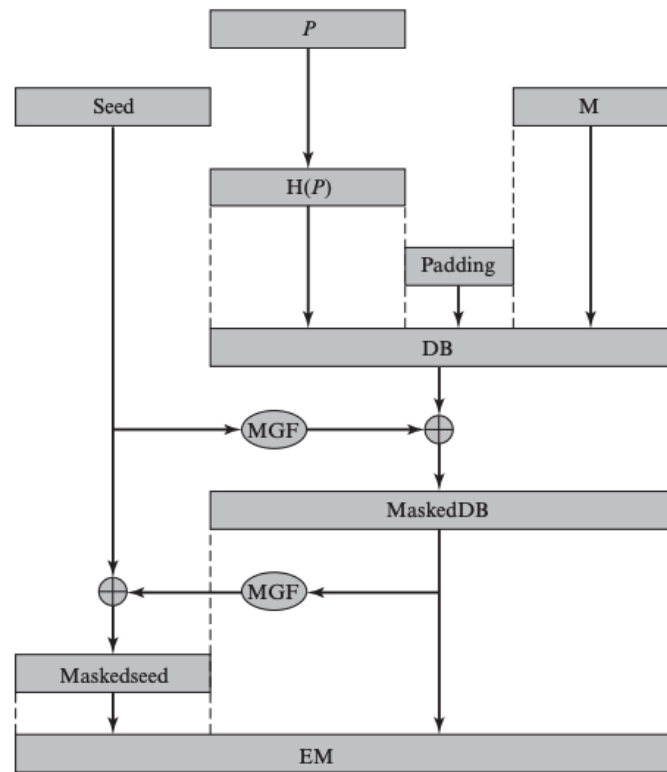


Figura 2.5: Estrutura do *Optimal Asymmetric Encryption Padding* [1].

totalidade. Para tal, recorre-se a vários tipos de estruturas, como por exemplo, *B-trees*, *heaps* e *hash tables* [10]. No entanto, para que tal aconteça, é necessário que os dados estejam em *plaintext* ou que a forma como foram cifrados não seja pseudo-aleatória.

A LOQR atualmente usa vários serviços PaaS (*Platform as a Service*) na *cloud* da Azure e da AWS (Amazon Web Services). A empresa compra estes serviços sique já têm alguma implementação de baixo nível, e onde se pode customizar o tipo de serviço oferecido. Desta forma, consegue através dessas plataformas criar máquinas virtuais em que consegue configurar aplicações *web* e o respetivo *back-end*. Essas máquinas virtuais são criadas através da plataforma do serviço *cloud*, sendo assim possível através de uma simples interface *web*, selecionar a customização pretendida (OS, CPU, RAM, etc.). No entanto, toda a infraestrutura por trás da máquina virtual é gerida pelo fornecedor de serviços *cloud*. Para além disso, a LOQR também adquire serviços SaaS (*Software as a service*), sendo que as suas bases de dados (SQL Server's) também são pré-configuráveis (espaço em disco, memória, etc.), mas já têm implementadas um *software* específico. De forma semelhante às máquinas virtuais, todo o *software* e *hardware* de suporte ao SQL Server é gerido pelo fornecedor de serviços *cloud*. Aos seus clientes fornece serviços SaaS, sendo que os clientes apenas interagem com aplicações de alto nível criadas pela LOQR. No contexto desta dissertação considera-se que todo o sistema da parte da LOQR funciona deste modo, estando tudo na *cloud*.

A conexão às bases de dados poderá ser feita através de uma API Java [11], facilitando a

incorporação da conexões num serviço *web* desenvolvido em Java. Para uma percepção mais fácil da estrutura da base de dados e dos dados presentes nela pode-se usar a ferramenta Azure Data Studio [12], onde apenas se necessita de estabelecer uma conexão à base de dados para depois enviar *queries* à base de dados, não sendo necessário desenvolver código, o que facilita processos mais dinâmicos (e.g. criação/eliminação de tabelas; Figura 2.6).

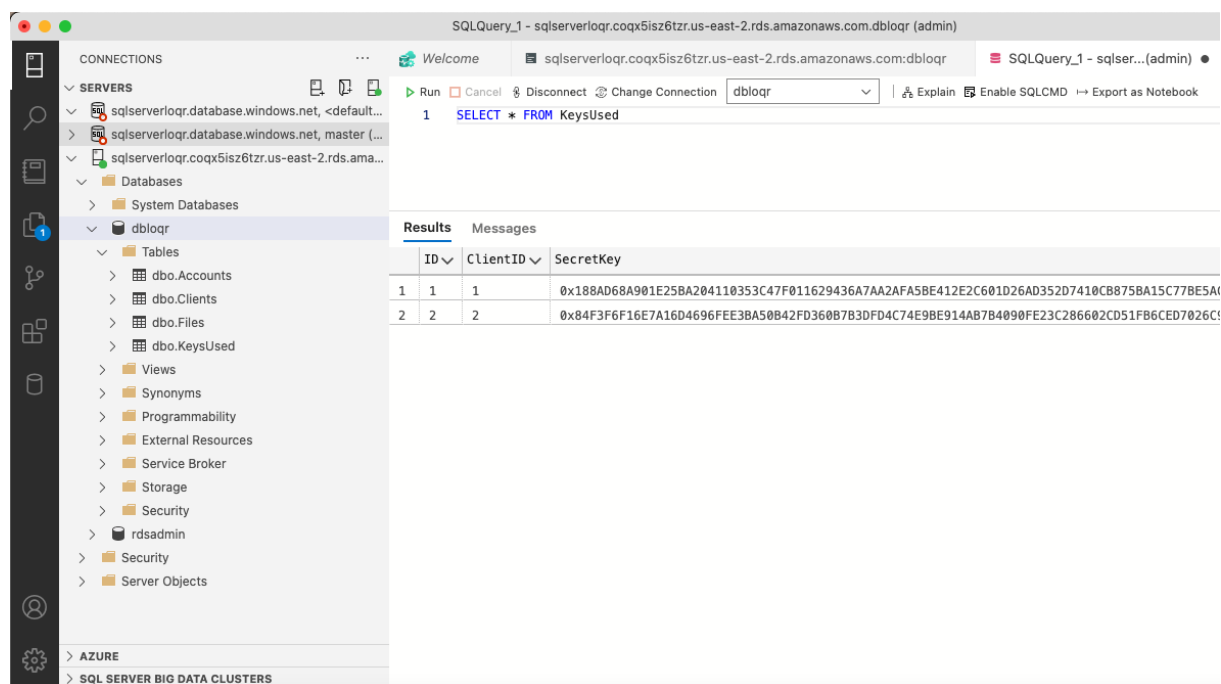


Figura 2.6: Exemplo de conexão e *query* no Azure Data Studio.

Surgem também em plataformas *cloud* serviços (SaaS) [13] [14] que facilitam aos seus utilizadores a criação, importação, rotação e eliminação de chaves criptográficas. Através destes serviços é possível ceder certas permissões de acesso a uma chave a utilizadores ou aplicações, sendo que a chave permanecerá durante o seu tempo de vida neste serviço. Como tal, uma das potencialidades deste serviço é o utilizador do serviço ceder um acesso restringido de uma chave a uma aplicação, e posteriormente negar esse acesso, ficando assim a aplicação impedida de usar a chave.

Estes tipos de serviços usam principalmente as primitivas criptográficas descritas na secção anterior, e serão demonstradas ao longo desta dissertação. Dado que não é possível verificar diretamente se essas primitivas estão corretamente implementadas, estes serviços têm de estar em conformidade com vários *standards* [15] [16].

Uma questão que também deve ser tomada em conta é a confiança que se pode ter nas implementações das primitivas criptográficas, que não pertencem a serviços de gestão de chaves, ou seja, pertencentes ao Java. Como tal, existe uma base de confiança em que o código implementado para estas primitivas é honesto e seguro, e corresponde aos *standards* que diz implementar [17]. A biblioteca do Java fornece todas as primitivas necessárias, e como tal, não foram usadas outras bibliotecas.

Também, é de esperar que os SQL Server's usados ao longo da implementação da solução proposta, estejam em conformidade com o que o fornecedor de serviços *cloud* diz fornecer, pois em causa está a sua reputação. Portanto, assume-se a disponibilidade dos dados lá colocados e restrição de acessos a estas bases de dados.

Capítulo 3

Trabalho Relacionado

Apesar de alguns dos trabalhos descritos neste capítulo não se apresentarem como uma solução direta para os objetivos que se pretendem alcançar nesta dissertação, estes trabalhos podem ter componentes semelhantes e que possam ser adaptados para a arquitetura futura da LOQR. A razão pela qual estes trabalhos não se apresentam como uma solução direta, deve-se ao facto de que estes propõem arquiteturas em que o cliente é quem interage diretamente com a base de dados, e portanto, só tem de confiar que os componentes no sistema de base de dados cumprem com as especificações de segurança que dizem possuir. Como tal, estes trabalhos não prevêm que exista um outro agente entre o cliente e a base de dados, que é o estado da arquitetura atual da LOQR. Apesar disso, estes trabalhos prevêm ataques contra sistemas em que as chaves criptográficas são mantidas temporariamente em memória.

Na secção 3.1 são descritos modelos de atacantes e falhas em bases de dados onde se pretende armazenar dados cifrados que permitam alguma indexação. Na secção 3.2 estão presentes três trabalhos semelhantes em que as suas aproximações passam por criar Bases de Dados que contenham um componente baseado em *hardware* na qual se possa confiar. Na secção 3.3 será descrito um trabalho em que existe um agente de confiança fora da base de dados e em que é possível definir vários níveis de segurança, que podem ser adaptados de acordo com pedidos feitos.

3.1 Modelização de Ataques

O trabalho de Benjamin Fuller et al.'s [18] surge porque não existem *standards* para o armazenamento de dados e várias opções baseadas em *software* vão surgindo, com componentes diferentes entre si. Pretende-se então fazer uma análise e comparação entre vários sistemas, algo que é difícil de se fazer, pois é necessário avaliar vários *tradeoffs*.

O desenho de um sistema de base de dados é um balanço entre segurança, funcionalidade, desempenho e usabilidade. Descrever a segurança de uma DBMS (*Database Management System*) passa por avaliar e classificar a informação que é revelada (*leaked*) a um atacante que tenha acesso

à base de dados. Funcionalidade corresponde ao tipo de *queries* que se pode fazer nesse sistema. Desempenho e usabilidade estão relacionados com os mecanismos de indexação, capacidade de computação, e a velocidade de tráfego na rede e da encriptação de dados.

Tais sistemas resultam sempre numa separação entre quem cria a *query*, e é também o criador dos dados, e quem processa a *query*, sendo este último, por vezes, uma entidade em que não se pode confiar, por vezes na *cloud*. Como tal, são considerados dois tipos de atacantes:

- *semi-honest*: um atacante que é honesto mas curioso, o que significa que vai seguir os protocolos mas que ocasionalmente tenta observar dados.
- malicioso: está disposto a executar qualquer ação para que consiga ler dados e/ou influenciar a execução do sistema.

Também são distinguidos atacantes que persistem durante grande parte do ciclo de vida da base de dados, e atacantes que apenas observam o estado da base de dados num determinado momento.

Em processos criptográficos, dependendo das operações criptográficas feitas e das primitivas usadas, pode existir informação a ser *leaked*. Como tal, os autores especificam um *leakage profile*, um *upper bound* da informação que é revelada à base de dados, para além da informação pretendida. Os parâmetros usados para definir estes perfis acabam por poder ser complicados, mas resumidamente podem caracterizar-se pelos objetos que são revelados, pelo tipo de informação que é revelado sobre esses objetos, que operação é que revela, e qual a parte do sistema que pode observar a *leakage*. Os seguintes tipos de objetos estão sujeito a *leakage*:

1. Dados guardados e estruturas de indexação.
2. *Queries*.
3. Respostas às *queries*.
4. Regras de controlo de acessos.

A *leakage* pode ser apenas resultante de uma *query* ou de um conjunto de *queries*, e pode variar dependendo das estruturas usadas para armazenar os dados, assim como do tipo das *queries* feitas. São então definidas cinco categorias de *leakage* relativas a uma objeto, ordenadas de ordem crescente com o nível de informação revelada, que podem acontecer durante o processo de inserção, pesquisa, ou alteração:

1. Estrutura: propriedades reveladas através do *padding* de um objeto, por exemplo, tamanho de uma *string*.
2. Identificadores: ponteiros para objetos que identifiquem as suas versões passadas ou futuras.

3. Predicados: Identificadores associados a informação adicional, por exemplo, corresponde a duas cláusulas numa *query*.
4. Igualdade: Objetos com o mesmo valor.
5. Ordem(ou mais): ordem numérica ou lexicográfica de objetos, ou até parte dos dados *plaintext*.

3.1.1 Cifras em Sistemas Legacy

Em sistemas *legacy* que se pretende manter as capacidades e optimizações de sistemas já implementados, é possível adicionar criptografia que preserve propriedades dos dados originais [18]:

- Igualdade: *Deterministic encryption* (DET) aplica uma permutação pseudo-aleatória mas determinística, que resulta em que igualdade de *ciphertexts* signifique igualdade de *plaintexts*, permitindo assim fazer pesquisas baseadas em igualdades de dados, mas não revelando **diretamente** qualquer outro tipo de informação sobre os dados. No entanto, cifrar dados deste modo revela igualdade de dados, sem que *queries* sejam feitas sobre esses dados.
- *Range*: *Order-preserving encryption* (OPE) preserva a ordem dos *plaintexts*, levando a que seja possível verificar a ordem a partir dos *ciphertexts*. No entanto, revelar a ordem acaba por ser considerado *leakage* e este modo também acaba por revelar alguns bits dos dados originais. Contudo, existem alguns métodos propostos que acabam por revelar menos informação.

3.1.2 Ataques a Dados Cifrados

Os modelos de ataque existentes [18] assumem que o atacante tem algum conhecimento inicial sobre dados armazenados ou *queries* realizadas. O sucesso do ataque está em obter informação para além da informação que já possui. Existem os seguintes tipos de informação inicial que um atacante pode ter:

- Conhecimento de toda a base de dados: o atacante só poderá tentar ter conhecimento de *queries*.
- Conhecimento de parte da base de dados: o atacante tenta obter informação sobre os dados restantes e sobre *queries*.
- Distribuição dos dados: o atacante pode ter conhecimento de informações estatísticas sobre os dados originais, por exemplo, a frequência de pessoas com um determinado nome num país.

- Distribuição das pesquisas: informação sobre a frequência de determinadas *queries*, de forma semelhante ao caso anterior.
- Conjunto de possibilidades: o atacante sabe que o dado pertence a uma conjunto limitado de dados, por exemplo, direções cardeais.

Os ataques que podem ser executados partem de dois princípios: diferentes *keywords* estão associadas a vários registos e parte dos sistemas revelam identificadores de *keywords* para registos armazenados (por exemplo, DET revela no processo de adicionar tabelas à base de dados se *keywords* forem iguais), e que dados podem ser comparados após várias pesquisas. Por exemplo, assumindo que um atacante tem conhecimento de uma *query* passada que continha "LastName = 'Smith'". Supondo também que o atacante tem acesso às seguintes *queries* "FirstName='John'" e "FirstName='Matthew'", que resultam em respostas com o mesmo número de registos, mas estes nomes estão cifrados e ele pretende diferenciar os dois. Para tal, um atacante sobrepõem os registos resultantes destas *queries* com a *query* que já tinha conhecimento prévio. Como o nome 'John Smith' é mais frequente do que 'Matthew Smith' o atacante consegue deduzir qual o primeiro nome que foi pesquisado. Este processo pode ser feito iterativamente para resultar em mais dados descobertos.

3.2 Computação Segura na Base de Dados

Os seguintes trabalhos apresentam todas soluções parecidas, em que a Base de Dados possui um componente em que se pode confiar, que opera sobre dados confidenciais que não são acessíveis ao resto do sistema.

Sumeet Bajaj e Radu Sion propõem o TrustedDB [19], um protótipo que parte do princípio que o *host* onde a base de dados reside não é de confiança, e portanto recorre ao uso de *hardware* de confiança presente nesse *host*. Este trabalho também tem um grande foco em justificar o uso de *hardware* de confiança, explicando que o seu uso acaba por ser mais barato do que o que é espectável. Para tal, é considerado um atacante curioso que tem como objectivo desrespeitar a confidencialidade do sistema. Ataques de *Denial of Service* estão fora do âmbito deste trabalho, assim como processos de autorização.

Neste protótipo foram usados os co-processadores criptográficos (SCPU), nomeadamente o IBM 4758 PCI e o IBM 4764 PCI-X. Estes componentes *tamper-resistant* são significativamente restringidos em memória e capacidade de computação, levando a que sistemas unicamente baseados nesses processadores sejam difíceis de implementar. Como tal, *queries* são pré-processadas para se identificar componentes sensíveis que tenham de ser computados dentro do SCPU, e *queries* com componentes não sensíveis serão computados no *host* não confiável. Apesar de *overheads* e limitações de desempenho, os autores afirmam conseguir custos mais baixos do que qualquer sistema baseado apenas em *software*, onde as *queries* são executadas em *hosts* nos quais não se confia, mesmo sendo pré-compiladas. Como custos associados à rede também devem ser tomados

em conta, selecionar um *sub-set* de dados para enviar ao cliente para que ele possa pesquisar por si próprio também não é uma boa opção. Este trabalho também tem como objetivo não limitar a possibilidade de expressão nas pesquisas feitas.

No modo principal de cifragem, parte dos dados são cifrados antes de processo de *outsourcing*. Os co-processadores presentes no *host* correm um sistema Linux embutido e podem ser programados personalizadas antes de serem integrados no sistema. Para processos criptográficos este suporta AES, 3DES (*Triple Data Encryption Algorithm*), chaves RSA até 4096 bits e funções *hash* criptográficas. Alternativamente a RSA, poderiam ser usadas assinaturas baseadas em ECC (*Elliptic-Curve Cryptography*), mas os autores do trabalho escolheram excluir esta opção, pois apesar de serem mais rápidas, acabam por não trazer um benefício em termos de custos. Como a maior parte das operações criptográficas são executadas no co-processador, um *hardware* desenhado para essas operações, para além de ser mais barato, acaba também por rápido. É pressuposto que existem métodos para proteção das comunicações (IPSec ou SSL; este trabalho foi desenvolvido antes do SSL ser descontinuado).

Fisicamente, o coprocessador conecta-se com outros componentes através de um PCI-X *bus*. O armazenamento de dados também é feito no *host*, e toda a base de dados será lá guardada. Se existirem páginas que estejam protegidas e que o SCPU pretenda aceder, estas são requisitadas *on demand* pelo *Paging Module* e consequentemente decifradas no interior do SCPU. Os atributos armazenados na base de dados podem ser privados ou públicos. Privados só podem ser decifrados pelo SCPU ou pelo cliente, dado que estes partilham a chave que foi usada para cifrar. Como foi dito anteriormente, computação de *queries* que envolvam dados públicos será feita pelo *host*, enquanto que a computação de dados privados é executada no SCPU. Os fluxos de processos são os seguintes e podem ser vistos na Figura 3.1:

0. Definição do esquema da base de dados. Também são assinalados os atributos considerados sensíveis com a *keyword* SENSITIVE:

```
CREATE TABLE customer(ID integer primary key,Name char(72) SENSITIVE, Address char(120) SENSITIVE);
```

1. A *query* é cifrada com a chave pública SCPU ou com uma chave simétrica gerada através de um acordo de chaves entre o SCPU e o cliente. Como tal, o *host* não consegue aceder à *query* em *plaintext*.
2. O *host* recebe a *query* e redireciona-a para o SCPU.
3. A *query* é decifrada pelo SCPU e é segmentada em *sub-queries* que contêm dados públicos e privados.
4. São redirecionadas para o *host* as *queries* públicas e para o motor de base de dados do SCPU as *queries* privadas.
5. O resultado final é agrupado, encriptado (e assinado se for desejado) e enviado para o cliente.

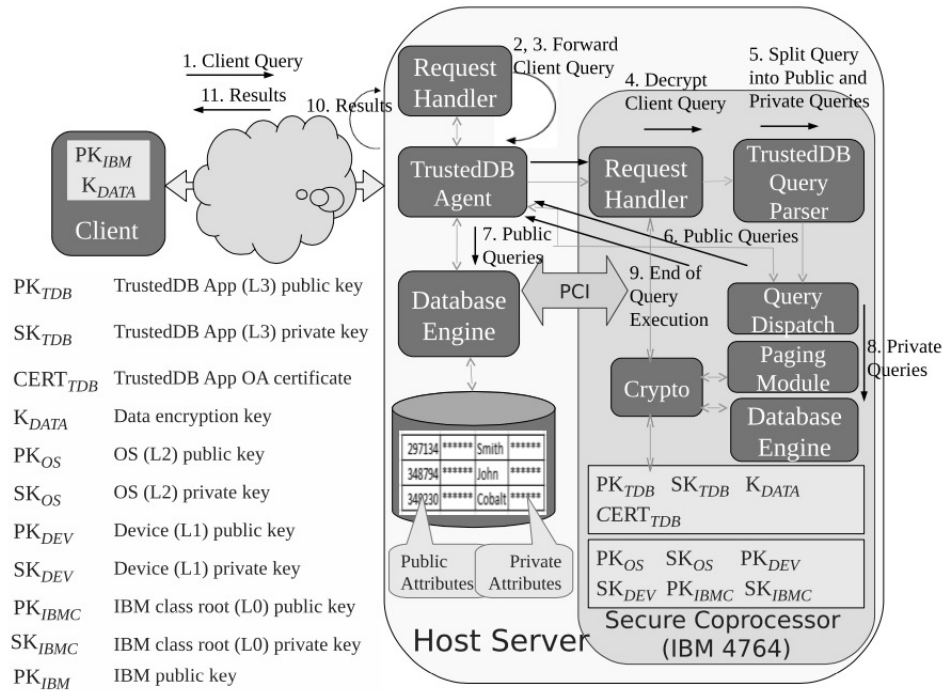


Figura 3.1: Arquitetura e sequência de operações da TrustedDB[19].

Como foi dito no parágrafo anterior, este esquema pressupõe a existência de uma chave compartilhada entre o cliente e o processador, sendo necessário confiar nas chaves presentes no SCPU. Pode-se assegurar que as chaves em memória não são acessíveis, pois este SCPU é *tamper-resistant*, estando em conformidade com FIPS (Federal Information Processing Standards) 140-2 nível 4 "Physical security requirements". Se for detectada qualquer tentativa de acesso não autorizado, pontos críticos da memória serão apagados, e as chaves não serão acessíveis.

Para se confirmar que o SCPU está a correr o código que é suposto correr, usa-se o SCPU *Outbound Authentication*. Resumidamente, este processo é feito através da construção de uma cadeia de certificados, onde camadas de *software* certificam a camada superior à sua. Na base desta cadeia de certificação está o certificado da chave pública do fornecedor, no meio estão certificados de camadas relativas ao processo de *boot* e do Sistema Operativo, e no topo desta cadeia encontra-se o certificado da aplicação que está a ser executada. Podendo, assim, um agente externo ao sistema verificar se essa aplicação está a ser executada no SCPU, requerendo esta mesma cadeia. *Hashes* do código presente em cada camada também são adicionadas aos certificados. O SCPU também autentica o cliente através da requisição da sua chave pública e certificado.

Neste sistema, também existem várias modos de cifrar os dados:

- A base de dados é cifrada e páginas cifradas são pedidas *on demand* pela TrustedDB. No entanto, como grande parte das operações têm de ser feitas pelo SCPU, esta opção limita o uso do CPU que não é de confiança e, portanto, perde-se capacidade computacional. Assim

sendo, é apenas aconselhado se todos os atributos forem considerados sensíveis.

- Apenas parâmetros considerados sensíveis serão cifrados usando AES, mas adicionando alguma aleatoriedade. Não é muito eficiente em termos de espaço, dado que alguns dados (p.e. inteiros) irão ter um *ciphertext* maior que o tamanho do *plaintext*.
- Igual ao anterior, mas é usada uma função de *hash* criptográfica para derivar uma chave para cada valor individual, através da chave usada anteriormente para cifrar todos os dados, e de dados adicionais sobre cada valor.

O CipherBase de Arasu et al.'s [20] teve como objetivo desenvolver um sistema (Figura 3.2) que adicionasse segurança a sistemas de bases de dados, mas que também preservasse as capacidades do SQL e de aplicações *legacy*, assim como permitir que o utilizador selecione o nível de segurança com que pretende proteger os seus dados. Este trabalho reconhece, desde o início, que não será uma solução para o futuro de segurança em bases de dados apenas encriptar os dados em armazenamento estático, dado que em processos de resposta a pedidos, é necessário descriptar parte da base de dados, necessitando assim de se ter acesso direto à chave, ficando ela, temporariamente, exposta em memória e ao CPU. Esta exposição acaba por adicionar vetores de ataque para um atacante.

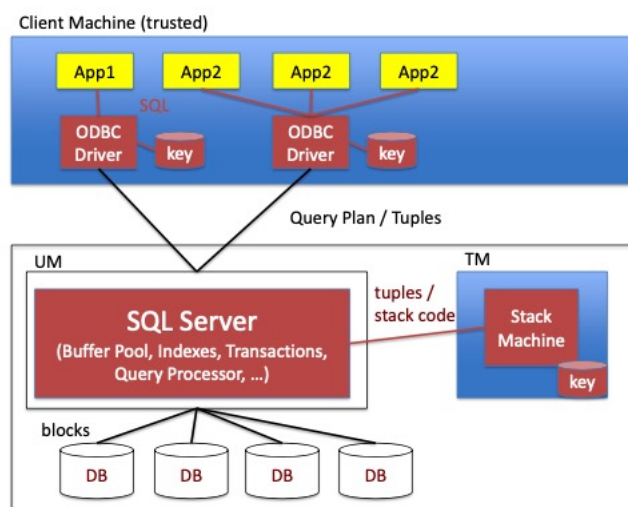


Figura 3.2: Arquitetura da CipherBase[20].

Pressupõe a existência de uma UM (*Untrusted Machine*), componente no sistema que é considerada como um possível atacante, podendo este ser um administrador da base de dados ou um atacante externo. É então adicionado o TM (*Trusted Module*), componente do sistema que vai ser responsável por uma grande parte de segurança, que recorre a um FPGA (Field-Programmable Gate Array), um circuito integrado onde o conjunto de operações que este executa e chaves criptográficas são embutidas em memória, durante o processo de fabrico. Isto reduz a superfície de ataque, pois o administrador da base de dados não consegue interagir com as operações que este executa, nem ter acesso à memória deste componente. A inacessibilidade a este componente representa a base de segurança para este sistema.

O cliente para interagir com o sistema usa uma consola genérica, e os acessos à base de dados são feitos através do *driver* ODBC (Open Database Connectivity), um componente que corre no ambiente seguro do cliente, e que é responsável por manipular e otimizar *queries*. Mas neste contexto, mais relevante por cifrar dados, constantes, definições de parâmetros de *queries* e de *updates*. A chave usada para cifração/decifração é uma chave AES de 128 bits pré-partilhada com o TM (a chave que é embutida em memória no processo de fabrico). É também neste componente que se tem de seleccionar quais as colunas que devem ser cifradas e qual o nível de segurança que se pretende para esses dados, podendo estes ser cifrados com AES-ECB se forem parâmetros que são únicos (por exemplo, nomes, *ids*), *order-preserving encryption* para poderem ser ordenados, ou AES-CBC para quando a confidencialidade destes é de muita relevância. Claramente, dados que não tenham qualquer importância associados a si próprios, serão mantidos em *plaintext*, no entanto, é preciso realçar que padrões de acesso a dados podem revelar informação relativamente a esses mesmos dados, mesmo que estes sejam cifrados.

Quando pedidos são enviados para a área da UM, a resposta tentará ser dada pela própria se forem respostas a que esta consiga responder, e que, portanto, não necessitem de operações criptográficas. O processamento de pedidos mais simples, sem recurso ao TM, vai adicionar pouco *overhead* às mensagens, manter a computação no seu local mais optimizado para tal e evitar trocas de mensagens desnecessárias com o TM. Quando existe a necessidade de recorrer a métodos criptográficos, os dados serão decifrados dentro do TM, computados e novamente cifrados, de modo a que possam ser enviados de volta para a UM, para chegarem ao cliente.

Também é possível cifrar a indexação de tabelas, armazenando na UM índices cifrados, com uma chave que não possui. Durante uma pesquisa que necessita de recorrer a indexação, que ocorre dentro do TM por apenas ser possível se os dados estiverem em *plaintext*, é necessário que sejam pedidas *pages* específicas à UM, enquanto se executa a travessia dos índices. Como o TM consegue decifrar as *pages*, consegue saber qual o índice que vai necessitar a seguir. Este modelo tem a vantagem de o armazenamento de índices cifrados ser numa zona sem confiança, mas leva a um aumento do número de mensagens que são trocadas.

Alguns anos depois de o TrustedDB ser publicado, surge o EnclaveDB de Christian Priebe et al.'s [21], um sistema que é de certa forma parecido ao seu antecessor, mas que dá uso a *hardware* mais recente. Pressupõe que a base de dados está presente num *host* que não é de confiança, e que está sujeito a ataques de administradores da base de dados ou de atacantes externos, e tenta ir mais longe que o CipherBase e o TrustedDB e garantir também confidencialidade de *queries*. Para alcançar isto, dados sensíveis são computados num enclave dentro do *host* que não é confiável (Figura 3.3), recorrendo a IntelSGX. O uso deste *hardware* protegido leva a que limitações discutidas anteriormente não sejam aplicáveis neste caso, pois, tabelas, índices, *queries* e outros metadados são guardados e processados na memória reservada para o enclave, sendo esta memória cifrada e também autenticada. Os autores justificam manter esses dados em memória dado que os preços de memória são cada vez mais baixos, levando a que um sistema deste tipo seja cada vez mais realista.

Mantendo-se dados na memória, que é reservada ao enclave, protege-se contra um atacante

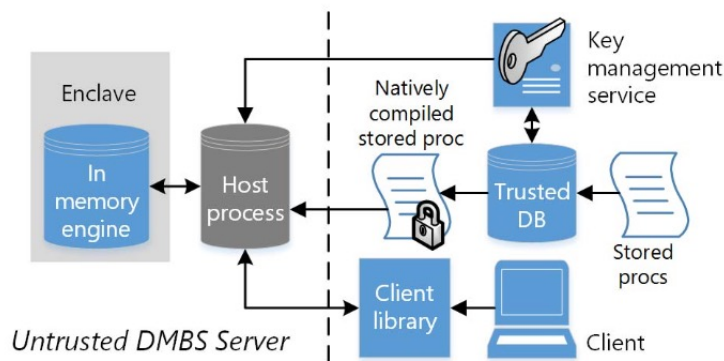


Figura 3.3: Arquitetura da EnclaveDB[21].

externo e contra aplicações que tenham privilégios superiores, considerando o próprio sistema operativo. Este é um ponto crítico deste sistema, pois mesmo um atacante que consiga comprometer o sistema operativo, não conseguirá comprometer a confidencialidade dos dados. É portanto considerado um atacante com elevadas capacidades computacionais, que consegue ler e modificar dados em todo o sistema, excluindo dentro do enclave. Os autores também reconhecem que não consideram, e que está fora do *scope* do trabalho, considerar ataques *side channel*.

Neste trabalho foi desenvolvido um protótipo usando o Hekaton, um motor de base de dados SQL Server que permite guardar algumas tabelas e índices em memória, e que vai guardando *checkpoints* em disco, para permitir a recuperação de dados após falhas. Quando uma aplicação cria um enclave no seu espaço de memória virtual, os dados e código presentes nesse enclave são guardados numa Enclave Page Cache (EPC), numa região da memória reservada ao processador. Como foi dito anteriormente, é garantida confidencialidade e autenticação dos dados, mas também são usadas *merkle trees* para garantir frescura e integridade dos dados. Bases de dados *legacy* recorrem a procedimentos do sistema operativo para gestão de *threads*, memória e armazenamento. Como neste sistema o próprio sistema operativo é considerado como um possível atacante, os processos de gestão de *threads* e memória são feitos por uma camada chamada *trusted kernel*, que proporciona ao motor Hekaton todos os serviços que precisa. Para armazenamento em disco pode-se recorrer ao sistema operativo num processo denominado *sealing*, em que o enclave guarda ou recupera dados confidenciais guardados no *host*, verificando também a sua integridade. Também foi desenvolvido um protocolo que garante integridade e frescura de dados e de *logs* da base de dados.

Assume-se que todos os componentes do lado do cliente são de confiança e que as suas chaves criptográficas estão corretamente protegidas, na mesma área que o cliente ou não. O cliente pode propor um desafio ao processador remoto de modo a que passe a confiar nele. A este processo chama-se *attestation*. O cliente consegue então verificar se o enclave foi iniciado no estado inicial que é suposto. Este sistema também suporta pré-compilação para código nativo numa máquina que o cliente confia, dentro da sua área. Para executar *queries* pré-compiladas, estabelece-se um canal seguro com o enclave, depois são feitos pedidos onde os parâmetros são cifrados, o enclave verifica a autenticação, decifra, executa as *queries* pré-compiladas, cifra os resultados da *query*

e depois envia a resposta para o cliente. O processamento de *queries* na memória do enclave permite menos interação entre o enclave e o *host*, e portanto, acaba por reduzir a superfície de ataque. Também é de notar que a pré-compilação e otimização de *queries* (num ambiente de confiança) separadas da execução da *query*, reduz também a superfície de ataque, dado que nestes processos pode existir alguma *leakage*. Por razões de desempenho, *queries* que operem sobre dados públicos, serão computadas pelo servidor que não é de confiança. No entanto, este sistema não suporta *queries* que operem em dados públicos e privados simultaneamente.

3.3 Pré-Processamento de Dados

O CryptDB de Popa et al.'s [22] apresenta-se como o trabalho que melhor se aproxima do estado atual do serviço da LOQR, dada a separação entre a cifra de dados e a base de dados. Este parte do princípio que num ponto crítico do processamento existe um servidor que não é de confiança. O objetivo é que o servidor que contém o DBMS compute dados encriptados, sem ter a necessidade de os decifrar, sendo estes apenas decifráveis por um agente de confiança. Para tal, recorre-se a várias maneiras de cifrar os dados. Está fora dos objetivos deste trabalho apresentar uma solução focado em integridade e disponibilidade, sendo a confidencialidade o ponto principal.

Neste sistema existem dois tipos de ameaças:

- Um atacante passivo que observa várias partes do sistema e que pretende obter informação através de informação cifrada, mas que não altera informação, podendo este ser um administrador da base de dados, um atacante que compromete o serviço da base dados ou quando a base de dados é *outsourced* e terceiros têm acesso aos dados. Tenta-se prevenir este ataque neste trabalho.
- Um atacante com controlo total sobre o DBMS. Existe proteção para utilizadores *logged-out*, mas não contra *logged-in*. O trabalho não se foca tanto neste ataque.

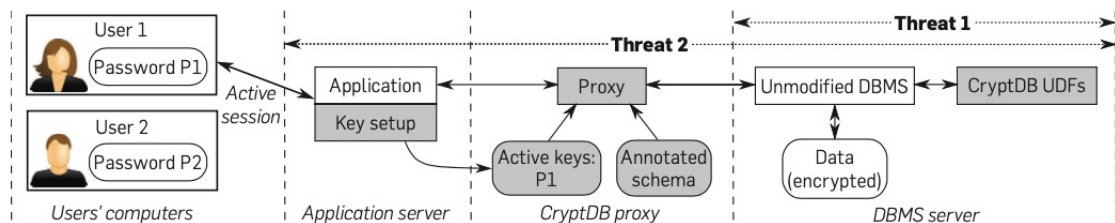


Figura 3.4: Arquitetura da CryptDB[22].

Como este sistema pretende uma integração fácil, aplicações já existentes não têm de ser alteradas, mas as *queries* têm de passar por um servidor *proxy* (Figura 3.4), um componente em que se confia. Este *proxy* guarda uma *master key* que usa para rescrever *queries* para que operem sobre dados encriptados. Para além de decifrar e cifrar dados, também altera alguns operadores

das *queries*, mas mantendo as suas semânticas. O servidor DBMS nunca tem acesso à *master key*, e portanto não consegue aceder ao *plaintext*, assegurando que um atacante passivo não tem acesso aos dados que pretende. O objetivo é que este servidor consiga processar *queries* como numa base de dados não cifrada, mas restringindo de modo a que este só compute as funções necessárias para conseguir processar a *query*. Para tal são usadas duas técnicas:

- *SQL-aware encryption*: dado que a maior parte das *queries* SQL são compostas por operadores básicos (somadas, agregadores, ordenamento), certos dados são cifrados através de esquemas adaptados para que esses operadores sejam suportados.
- *Adjustable query-based encryption*: ajusta o esquema para que suporte vários operadores. Para tal, *ciphertexts* são guardados dentro de outros *ciphertexts*, criando uma espécie de *onions*. Desta maneira esquemas mais inseguros não serão revelados para dados em que não seja preciso usar tais esquemas.

Adicionalmente, também é adicionada confidencialidade para nome de tabelas e colunas, mas não para as restantes partes da base de dados (por exemplo, relações entre tabelas). Para proteger contra um atacante passivo, em vez de se armazenar a *master key* no servidor, usa-se uma chave derivada através da password de autenticação do cliente. Portanto, a chave estará segura contra um atacante passivo apenas se o utilizador estiver *logged-out* e registos de acesso já não estiverem em memória. O *proxy* guarda o esquema da base de dados e informações relativas à cifra das colunas. O DBMS armazena identificadores de tabelas e colunas, os dados cifrados e tabelas auxiliares. O DBMS também possui UDF (*user-defined functions*) que permitem ao servidor computar os dados cifrados. Para processar uma *query* existem os seguintes passos:

1. A Aplicação envia uma *query* que passa pelo *proxy* e é reescrita, alterando o nome da tabela, o nome das colunas e as constantes. As constantes são cifradas com um determinado esquema, dependendo da forma que se pretende operar esses dados. É usada a *master key* para derivar outras chaves (detalhado mais à frente) que serão usadas para cifrar os dados.
2. O *proxy* verifica se os dados estão na forma correta para as *queries* serem executadas. Se não estiverem, o *proxy* envia a chave necessária para ser aplicada uma UDF no servidor DBMS, para que este consiga processar os dados.
3. O *proxy* envia a *query* para o servidor DBMS.
4. O servidor envia os resultados que serão decifrados no *proxy* e redirecionados para o cliente.

Para cifrar os dados existem os seguintes métodos:

- *Random* (RND): Este método é o mais seguro, mas não permite que sejam executadas computações eficientes sobre os dados. Usa *Blowfish* ou AES em modo CBC com um IV aleatório, o que leva a que dois *plaintexts* iguais resultem em dois *ciphertexts* diferentes.

- *Deterministic* (DET): Permite ao servidor executar operações de igualdade, pois igualdade de *ciphertexts* resulta em igualdade de *plaintexts*. Para tal é usado Blowfish ou AES em modo CMC.
- *Order-preserving encryption* (OPE): Neste método de cifragem os dados são cifrados de modo a que os *ciphertexts* preservem a ordem dos *plaintexts*. Como acaba por revelar alguma informação sobre os dados originais, este método apenas é usado em colunas que o cliente seleciona.
- *Homomorphic Encryption* (HOM): Funciona de forma semelhante ao RND mas permite que o servidor DBMS compute dados cifrados que poderão ser decifrados no servidor *proxy*. Baseado no sistema criptográfico de Paillier [23], em que multiplicar a cifra de dois dados resulta na cifra da soma desses mesmos dados. Para tal, operações, por exemplo SUM, são substituídas por UDF's no servidor *proxy*, para que o servidor DBMS consiga computar corretamente o resultado pretendido.
- *Join* (JOIN and OPE-JOIN): No esquema DET são usadas chaves diferentes para colunas diferentes, para prevenir correlação entre colunas. Portanto, é definido um esquema que permite manter valores iguais entre colunas. OPE-JOIN possibilita uniões através de relações de ordem.
- *Word search* (SEARCH): Usado como uma alternativa cifrada à operação LIKE. O *proxy* gera um *token* que representa a cifra de um determinado texto, e o DBMS consegue determinar se existe algum texto cifrado que corresponda ao *token*. No entanto este esquema apenas suporta pesquisas por palavras "completas".

Foi ainda, criado o esquema *Adjustable Query-Based Encryption*, para maximizar a segurança encriptando os dados de forma a que seja aplicado o esquema mais seguro possível, dependendo das *queries* que se pretendem executar nesses dados. Por exemplo, se sobre uma determinada coluna não se pretende realizar igualdades, então deverá ser usado o esquema RND, caso contrário, usa-se o esquema DET, mas não OPE. No entanto, o tipo de *query* que vai ser executado sobre os dados pode não ser conhecido inicialmente. Para resolver este problema, o CryptDB ajusta dinamicamente o nível de cifra usado, criando *onions* (Figura 3.5), estruturas em que os dados são encriptados em várias camadas, que permitem níveis de computação e de segurança diferentes. Vários tipos de *onions* são necessários, dado que há esquemas usados para diferentes funções, e cada coluna não tem de ter todos os tipos de *onions*. Para cada camada dos *onions*, pertencentes a uma determinada coluna, é usada a mesma chave, mas entre colunas diferentes e *onions* de colunas diferentes as chaves não são partilhadas. Estas chaves são derivadas da MK, da seguinte forma:

$$K_{t,c,o,l} = \text{PRP}_{MK}(\text{table } t, \text{column } c, \text{onion } o, \text{layer } l),$$

onde PRP representa uma permutação pseudo-aleatória, *t* uma tabela, *o* uma *onion* e *l* uma camada dessa *onion*.

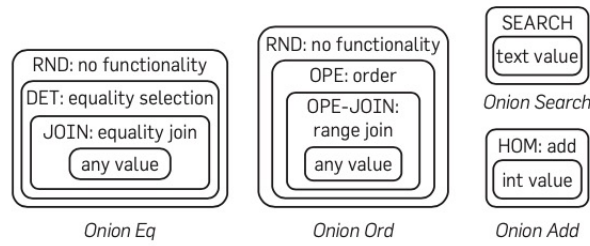


Figura 3.5: Onions que permitem vários níveis de encriptação[22].

Todos os dados são cifrados inicialmente até à camada mais segura, e camadas vão sendo removidas de acordo com os pedidos feitos pelo cliente (Figura 3.6). Se um dado tem de ser computado numa camada em que não está, e o cliente pretende que esse dado seja computado dessa forma, o servidor *proxy* envia a chave usada para essa camada para que o servidor DBMS consiga obter o resultado. É de notar que as chaves são independentes entre camadas, e que a função de derivação da chave não é reversível, e portanto, ceder uma chave, não compromete outras. Os dados nunca serão decifrados para além da camada menos segura, o que levaria a que os dados fossem obtidos em *plaintext* no lado do servidor DBMS, mas também é possível configurar para que não se removam camadas a partir de uma determinado nível e que apenas se usem determinados modos (por exemplo, sobre dados de cartões de credito podem ser removidas camadas de RND para DET, mas nunca deve ser usada OPE).

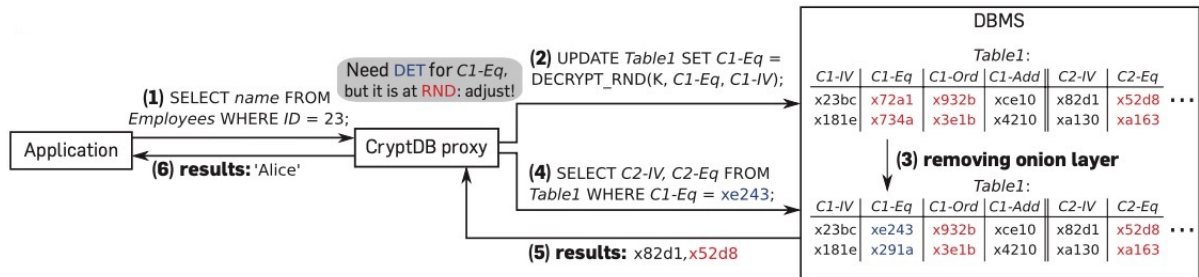


Figura 3.6: Cliente faz um pedido em que é necessário remover uma layer[22].

No entanto, JOINS de igualdade são facilmente implementados se desde o início se souber que é suposto fazer tal em colunas específicas. Simplesmente se usa a mesmas chaves para DET, em colunas em que se quer fazer JOIN. Para ser possível fazer JOIN em casos que não se sabe inicialmente, é adicionada a primitiva JOIN-ADJ (*adjustable join*), que permite que quando se pretende fazer o JOIN de duas colunas diferentes, através de requisição de chaves, remove-se uma camada dos *onions*, de modo a que fiquem em camadas onde é possível comparar *ciphertexts* iguais. OPE-JOIN só é suportado se for requerido antes.

Capítulo 4

Análise de Requisitos

Neste capítulo será especificado o estado da arquitetura actual da LOQR para armazenamento seguro dos dados, assim como das tecnologias que estão em uso para esse efeito. A secção 4.1 explica resumidamente duas tecnologias usadas para cifrar os dados, a secção 4.2 desenvolve várias partes do sistema que devem ser analisadas para entender o serviço atualmente fornecido e a secção 4.3 detalha os requisitos que se pretendem alcançar na futura solução.

4.1 Tecnologias Integradas

Existem duas tecnologias que são usadas pela LOQR que representam uma parte bastante relevante no âmbito deste trabalho, e como tal serão descritas singularmente. Para se entender o papel destas no sistema, é necessário entender a base delas. Estas vão ser descritas nas duas próximas subsecções.

4.1.1 Always Encrypted

Esta tecnologia[24] surge com a intenção de proteger dados sensíveis armazenados em servidores SQL da Microsoft. É possível cifrar dados do lado do cliente de modo que a base de dados em que estes estão armazenados, não tenha acesso à chave que foi usada para cifrar. Esta separação entre quem pode aceder aos dados e quem gere o armazenamento dos dados, acaba por reduzir a superfície de ataque.

Uma outra grande vantagem deste *software* é que a cifra dos dados acaba por ser praticamente transparente para o utilizador. Uma *driver* instalada no computador do cliente automaticamente decifra/cifra dados considerados sensíveis, sem que a aplicação faça essa função. Ou seja, dados em colunas seleccionadas como sensíveis são automaticamente cifrados antes das *queries* serem enviadas para a base de dados, e os dados cifrados obtidos em respostas, também são automaticamente decifrados antes de chegarem à aplicação.

Para tal, as colunas que contêm dados sensíveis têm de ser selecionadas pelo utilizador. É também selecionado o algoritmo usado para cifrar os dados e quais as chaves CEK (*column encryption key*) usadas para cifrar essas colunas. Para além dessas chaves, também é usada uma chave mestra CMK (*column master key*) para proteger essas chaves.

O motor da base de dados apenas guarda metadados sobre a configuração de cifra de colunas sensíveis. Isso inclui armazenamento cifrado das CEK. Podem ser armazenadas duas cifras da mesma CEK, cifradas com CMKs diferentes, para que o processo de rotação de CMKs seja mais fácil. O armazenamento da CMK também pode ser escolhido pelo utilizador. Pode-se usar soluções da Microsoft ou escolher o seu próprio *provider*.

Quando o *driver* pretende processar uma *query*, este contacta a *key store* para obter a CMK e assim conseguir decifrar a CEK que está cifrada. O *plaintext* da CEK pode ser armazenado em *cache*, para reduzir o número de acessos feitos.

Como foi visto no capítulo 3, a forma como os dados são cifrados vai ter impacto nas operações que podem ser feitos nestes. O Always Encrypted suporta dois modos de encriptar os dados:

- Determinístico: dados iguais irão resultar em *ciphertexts* iguais. Isto permite fazer algumas operações nesses dados (operações de igualdade, JOIN). Mas como foi referido no capítulo 3, é possível retirar informações sobre este tipo de dados.
- Pseudo-Aleatório: dados iguais resultam em *ciphertexts* diferentes. Este método é mais seguro, mas não permite operações sobre estes dados. Dados em que não se pretende realizar qualquer tipo de operação deverão ser cifrados aleatoriamente.

4.1.1.1 Processo de Cifra

As CMKs cifram CEKs usando RSA com *Optimal Asymmetric Encryption Padding* (RSA-OAEP), um esquema RSA que adiciona alguma aleatoriedade aos dados.

Para cifrar dados [25] o Always Encrypted usa o algoritmo AEAD_AEAD_AES_256_CBC_HMAC_SHA_256 que é um esquema do tipo *Encrypt-then-MAC*. Para cifrar um dado são executados os seguintes passos:

1. A forma como o IV (*Initialization vector*) é gerado, depende de como se quer cifrar os dados.
 - Para dados cifrados aleatoriamente, o IV é gerado aleatoriamente, levando a que dados iguais resultem em cifras diferentes:
`IV = Generate cryptographically random 128 bits`
 - Para dados cifrados de uma forma determinística, o IV é gerado através do valor *plaintext*, resultando em *ciphertexts* iguais:
`IV = HMAC-SHA-256( iv_key, cell_data ) truncated to 128 bits,`
 onde *iv_key* é derivado de CEK da seguinte forma:

- ```
iv_key = HMAC-SHA-256(CEK, "Microsoft SQL Server cell IV key"+
algorithm + CEK_length).
```
2. Depois, é gerado o *ciphertext*: `aes_256_cbc_ciphertext = AES-CBC-256(enc_key, IV, cell_data)` with PKCS7 padding, onde `enc_key` é derivada de CEK da seguinte forma:
- ```
enc_key = HMAC-SHA-256(CEK, "Microsoft SQL Server cell encryption
key"+ algorithm + CEK_length )
```
3. A partir do *ciphertext* é gerado o MAC(Message Authentication Code):
- ```
MAC = HMAC-SHA-256(mac_key, versionbyte + IV + Ciphertext +
versionbyte_length)
```
- onde
- ```
versionbyte = 0x01 and versionbyte_length = 1
```
- e,
- ```
mac_key = HMAC-SHA-256(CEK, "Microsoft SQL Server cell MAC key"+
algorithm + CEK_length)
```
4. Depois o resultado final é produzido através da concatenação de um *byte* contendo a versão do algoritmo usado, o MAC, o IV, e o *ciphertext*:
- ```
aead_aes_256_cbc_hmac_sha_256 = versionbyte + MAC + IV +
aes_256_cbc_ciphertext
```

O resultado final acaba por ter um tamanho acrescido, dado que os seguintes componentes têm os seguintes tamanhos (em bytes):

- versionbyte: 1
- MAC: 32
- aes_256_cbc_ciphertext: $(\text{FLOOR}(\text{DATALENGTH}(\text{cell_data}) / \text{block_size}) + 1) * \text{block_size}$, onde:
 - *block_size* são 16 bytes
 - *cell_data* representa o valor *plaintext* de um dado

Portanto, o tamanho final de um *ciphertext* resultante da cifra de um valor *plaintext*(*cell_data*) será:

$$1 + 32 + 16 + (\text{FLOOR}(\text{DATALENGTH}(\text{cell_data}) / 16) + 1) * 16.$$

Para *plaintexts* relativamente grandes (p.e. 2000 bytes) o tamanho não é muito acrescido, mas para *plaintexts* mais pequenos acaba por aumentar bastante o tamanho de armazenamento. Por exemplo, um *plaintext int* de 4 bytes acaba por ter um *ciphertext* de 65 bytes.

4.1.2 S3

Na secção anterior foi explicada a tecnologia que é usada para cifrar certas colunas em tabelas de uma base de dados. Nesta vai ser descrito o processo de cifra [26] no S3 (*Simple Storage Service*), um tipo de armazenamento que a LOQR usa para guardar ficheiros binários (objetos). Para tal, é usado *Client-side encryption*, de forma semelhante ao Always Encrypted, sendo que é um processo transparente para o utilizador.

Para tal o utilizador apenas tem de seleccionar no *client* (aplicação usada para comunicar com a *Cloud*) a localização de uma chave mestra a usar. Esta chave pode ser uma chave simétrica ou um par de chaves assimétricas.

Quando um utilizador pretende fazer *upload* de um objeto acontecem os seguintes passos:

1. O *client* S3 gera uma chave simétrica DEK (*data encryption key*), que será usada apenas para esse objeto, e cifra o objeto.
2. O *client* cifra a DEK usando uma chave mestra e faz *upload* da cifra da chave em conjunto com outros metadados. O *client* pode depois saber qual a chave mestra que deve usar para decifrar.
3. O *client* faz upload do objeto cifrado e dos seus metadados (contém a DEK cifrada).

Quando o utilizador faz *download* de um objeto, sabe pelos metadados qual a chave mestra que foi usada para cifrar a DEK. Consegue então decifrar a DEK cifrada, e depois usá-la para decifrar o objeto.

Para a cifra de objetos é utilizado AES-GCM, que suporta chaves de tamanho 128, 192 e 256, com um vetor de inicialização de 96 *bits*. Também é possível gerar uma *tag* de autenticação recorrendo a ECDSA.

4.2 Especificação da Arquitetura Atual

O objectivo deste secção é especificar onde reside a segurança do serviço que actualmente a LOQR oferece ao seus clientes, para ser possível entender a mudança que tem de ser feita para alcançar os objetivos pretendidos. Vão então ser abordados os atores que participam no sistema e quais as suas funções, a arquitetura e os respectivos componentes, o fluxo de dados nesses mesmo componentes, os ativos e o modelo de confiança.

4.2.1 Agentes

Nesta secção pretende-se identificar os agentes presentes no serviço oferecido pela LOQR.

- Cliente - Este foi quem comprou o serviço à LOQR. A única função deste é interagir com o *browser* que comunica com o serviço. Toda a arquitetura, manutenção, gestão de chaves, funcionamento da base de dados e comunicações são invisíveis para ele. Também existem clientes que têm a capacidade de aceder a serviços de *back office*.
- Operador da LOQR - Tem que garantir que os pedidos de acesso à informação requerida pelo Cliente são respondidos. É também responsável por criar, alterar e apagar as chaves usadas para cifrar/decifrar os dados que estão presentes nas bases de dados, e de disponibilizar o *software* disponível para tal.
- Administradores Cloud - Têm a responsabilidade de manter disponível os serviços onde são hospedados as bases de dados.
- Administrador ISP - Apenas certifica que as comunicações entre os vários participantes estão disponíveis.

4.2.2 Arquitetura SQL Server

Aqui vão ser detalhados os componentes e os fluxos de dados do serviço que integra o SQL Server da Microsoft.

4.2.2.1 Componentes

A Figura 4.1 representa o serviço principal oferecido pela LOQR. A área “Instalações Cliente” representa as instalações do cliente, a área “Instalações LOQR” é composta por várias VM’s em instâncias diferentes também na Cloud que tem os seus componente replicados, e a área “Cloud” representa serviços na Azure. É composto pelos seguintes componentes:

- Utilizador Final - Este componente representa a aplicação desenvolvida pela LOQR que pode ser acedida através do *browser* dos clientes, não sendo necessária qualquer gestão por parte dos clientes. Este contém várias funcionalidades, mas nenhuma relacionada com segurança, sem ser a de comunicar os dados ao servidor da LOQR, que é feita através de HTTPS. É através deste componente que os clientes interagem com o sistema.
- Operador Cliente - Atualmente, este componente apenas representa a interação existente com serviços de *backoffice*, mas poderá ter mais funcionalidades na futura arquitetura.
- Aplicação LOQR - Este servidor opera como intermediário entre o cliente e a base de dados SQL. A sua função é cifrar *queries* do cliente e transmiti-las para a base de dados da Microsoft e decifrar a resposta obtida relativamente à *query*, e transmiti-la ao cliente. O processo de cifrar *queries* e decifrar respostas é feito através do *driver* Always Encrypted, usando uma chave gerada no KeyManager, estando estes dois componentes presentes no servidor (no esquema são apresentados separadamente para uma melhor percepção).

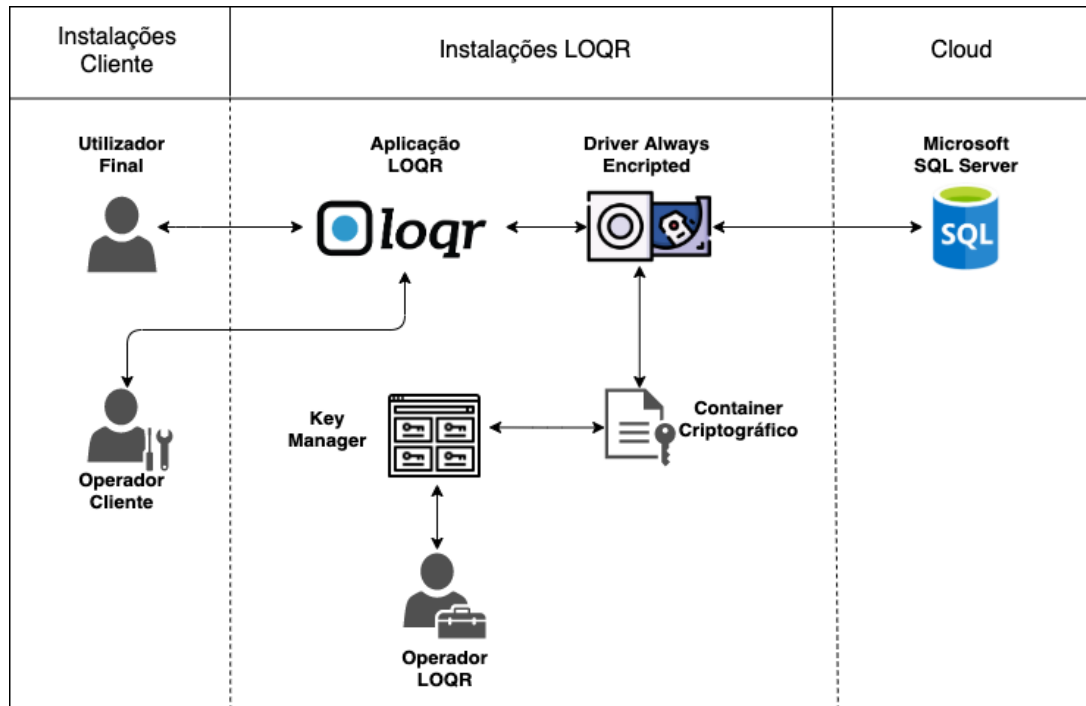


Figura 4.1: Arquitetura do serviço de armazenamento que integra o SQL Server.

- Operador da LOQR - Este apenas representa o operador da LOQR que interage com o KeyManager.
- KeyManager - Neste componente da arquitetura, o Operador da LOQR pode criar, alterar e eliminar chaves simétricas que serão usadas pelo *driver* Always Encrypted. Para a criação de chaves é usado o Java keytool. No entanto, as chaves durante o seu ciclo de vida irão ficar guardadas no Container.
- Container Criptográfico - Este componente é um ficheiro P12 que armazena chaves criptográficas. Este ficheiro é protegido através de uma *password*. Aqui são armazenadas as chaves mestras de cada cliente.
- *Driver* Always Encrypted - Quando existe uma *query* para ser feita à base de dados, este *driver* automaticamente cifra colunas da *query*, sendo este mecanismo invisível para a aplicação e automatizado. Apenas têm de ser escolhidas as colunas que devem ser encriptadas e quais as chaves a usar. Para acesso à chave, este acede ao Container Criptográfico.
- Servidor SQL MS - Esta base de dados é criada e administrada pela LOQR. Esta recebe os dados enviados pela LOQR e armazena-os.

4.2.2.2 Fluxos de Dados

A Figura 4.2 representa o fluxo de dados para este serviço, contendo a seguinte legenda:

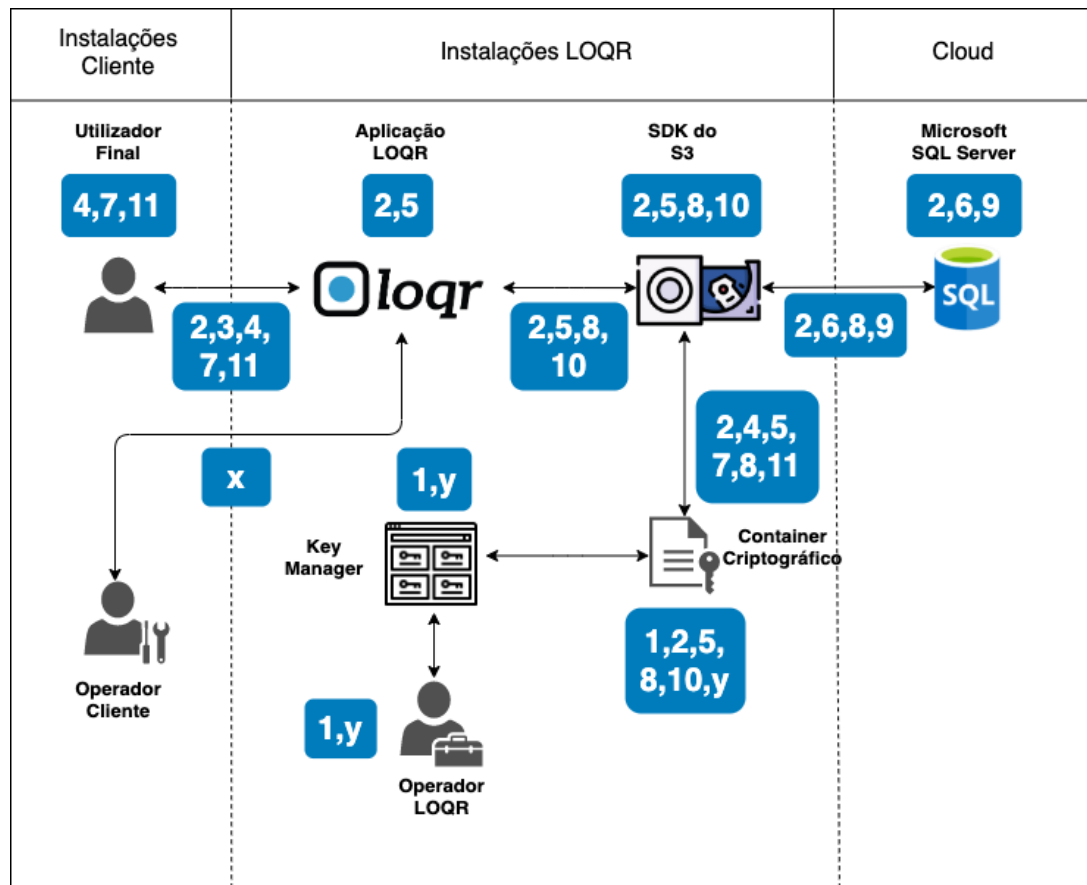


Figura 4.2: Fluxo de dados do serviço de armazenamento que integra o SQL Server.

1. Antes de qualquer comunicação, é criada no KeyManager uma chave mestra para o cliente (única para cada cliente), no entanto, esta nunca lhe vai ser enviada e vai ficar presente no Container, durante toda a sua existência. É a única chave gerada pelo KeyManager para este serviço.
2. O operador da LOQR vai garantir que a base de dados da Microsoft está disponível para guardar dados, e que os outros componentes estão corretamente configurados. O *driver* Always Encrypted vai ser configurado para usar a chave guardada no Container.
3. Quando tudo estiver configurado e a aplicação instalada do lado do cliente, são definidos os parâmetros para comunicações e algoritmos de criptografia a usar entre a aplicação cliente e o servidor da LOQR. A partir deste passo, assume-se a disponibilidade do servidor da LOQR, assim como de todas as comunicações existentes, incluindo também a disponibilidade do servidor SQL.
4. O cliente está a usar a aplicação e pretende adicionar dados novos à base de dados. A *query* é então enviada ao servidor da LOQR para que seja possível obter a resposta.
5. A *query* antes de ser redirecionada para o servidor SQL, passa primeiro pelo *driver*, onde as colunas seleccionadas anteriormente, serão cifradas. Se a chave não se encontrar na cache da Drive, é necessário aceder ao Container.

6. A *query* cifrada é enviada para o SQL Server, onde será processada e os dados armazenados. Também são guardadas evidências de que o processo foi corretamente executado.
7. O cliente está a usar a aplicação e pretende aceder a dados presentes na base de dados. Uma *query* é enviada ao servidor da LOQR contendo os dados que pretende obter.
8. A *query* antes de ser redirecionada para o servidor SQL, passa primeiro pelo *driver*. Se colunas que contêm dados privados, serão cifradas. Se a chave não se encontrar na cache da Drive, é necessário aceder ao Container.
9. A *query* é processada pela base de dados e enviada ao servidor da LOQR.
10. Após se obter a resposta da base de dados, o *driver* decifra as colunas com dados cifrados. Mais uma vez, se a chave não se encontrar na cache da Drive, é necessário aceder ao Container.
11. A resposta à *query* é enviada e apresentada ao cliente.
- x. Durante os processos anteriores, o Operador Cliente poderá aceder a serviços de *back office*, sem qualquer ordem específica.
- y. O Operador da LOQR em qualquer momento poderá aceder ao KeyManager e criar, editar ou eliminar chaves.

4.2.3 Arquitetura AWS

Aqui vão ser detalhados os componentes e os fluxos de dados do serviço que integra o Armazenamento S3.

4.2.3.1 Componentes

A Figura 4.3 representa a arquitetura do serviço de armazenamento de ficheiros binários. A área “Instalações Cliente” representa as instalações do cliente, a área “Instalações LOQR” é composta por várias VM’s em instâncias diferentes também na Cloud que tem os seus componentes replicados, e a área “Cloud” representa serviços na AWS. Podemos verificar a existência dos seguintes componentes:

- Utilizador Final - Este componente representa a aplicação desenvolvida pela LOQR que pode ser acedida através do *browser* dos clientes, não sendo necessária qualquer gestão por parte dos clientes. Este contém várias funcionalidades, mas nenhuma relacionada com segurança, sem ser a de comunicar os dados ao servidor da LOQR, que é feita através de HTTPS. É através desta componente que os clientes interagem com o sistema.
- Operador Cliente - Atualmente, este componente apenas representa a interação existente com serviços de *backoffice*, mas poderá ter mais funcionalidades na futura arquitetura.

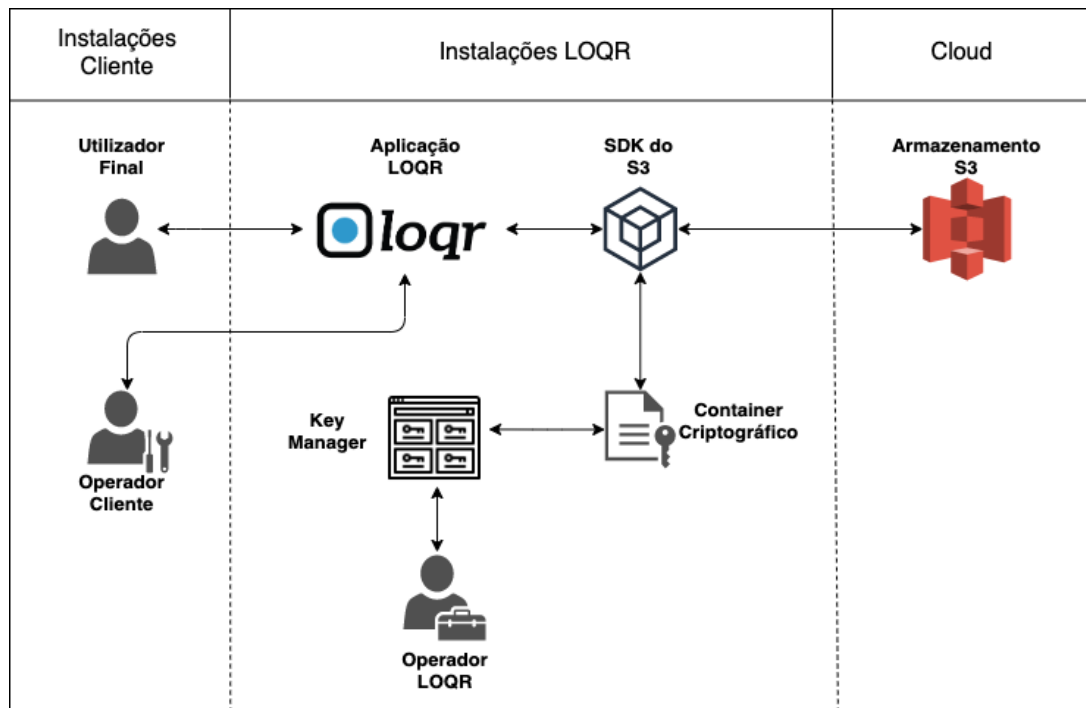


Figura 4.3: Arquitetura do serviço de armazenamento que integra o S3.

- **Aplicação LOQR** - Este servidor opera como intermediário entre o cliente e a base de dados. A função deste é cifrar objetos e adicioná-los à base de dados e receber *queries* sobre objetos presentes na base de dados e devolvê-los ao cliente. O processo de cifrar/decifrar objetos é feito através do SDK do S3, usando uma chave que é gerida pelo KeyManager, estando estes dois componentes presentes no servidor (no esquema são apresentados separadamente para uma melhor percepção).
- **Operador da LOQR** - Este apenas representa o operador da LOQR que interage com o KeyManager.
- **KeyManager** - Neste componente da arquitetura o Operador da LOQR pode criar, alterar e eliminar chaves simétricas que serão usadas pelo SDK do S3. Para a criação de chaves é usado o Java keytool. No entanto, as chaves durante o seu ciclo de vida serão guardadas no Container.
- **Container Criptográfico** - Este componente é um ficheiro P12 que armazena chaves criptográficas. Este ficheiro é protegido através de uma *password*. Aqui são armazenadas as chaves mestras de cada cliente.
- **SDK do S3** - Utilitário criado pela Amazon que pode ser integrado em aplicações dos seus próprios clientes (p.e. no servidor da LOQR). O seu objectivo é decifrar/cifrar objectos com uma chave AES de 256 *bits* gerada para cada objeto. A chave usada para cifrar o objeto, é também cifrada com a chave mestra presente no Container, e armazenada em conjunto com o objeto cifrado na base de dados. Quando um objeto é recebido da base de dados, o SDK decifra a chave cifrada e prossegue para a decifração do objecto, necessitando

de ter acesso à chave mestra. Este componente não armazena a chave mestra mas precisa de ter uma referência do local onde está armazenada.

- Base de dados S3 - Esta base de dados é criada e administrada pela LOQR no serviço AWS S3, sendo hospedada nos serviços Cloud da Amazon. Esta recebe os dados enviados pela LOQR e armazena-os.

4.2.3.2 Fluxos de Dados

A Figura 4.4 representa o fluxo de dados para este serviço, contendo em baixo a legenda.

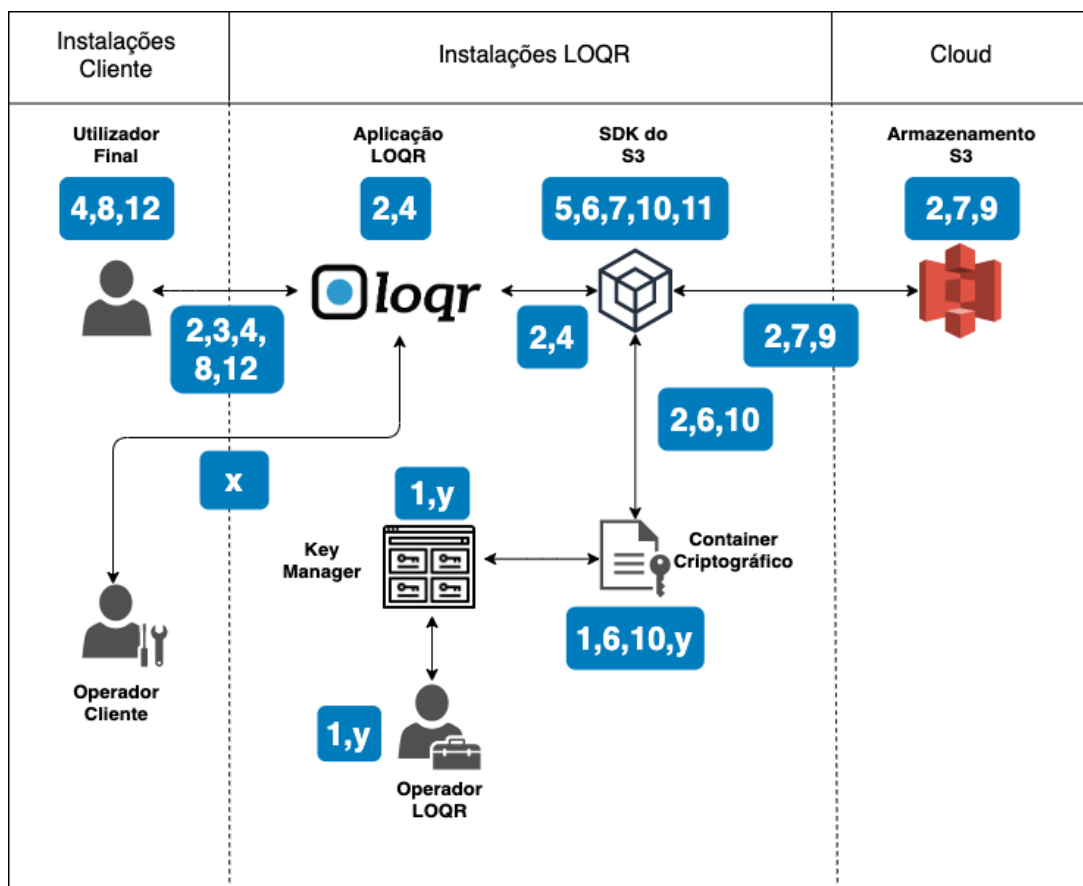


Figura 4.4: Fluxo de dados do serviço de armazenamento que integra o armazenamento S3.

1. Antes de qualquer comunicação, é criada no KeyManager uma chave mestra para o cliente (única para cada cliente), no entanto, esta nunca lhe vai ser enviada e vai ficar presente no Container, durante toda a sua existência.
2. O operador da LOQR vai garantir que a base de dados S3 está disponível para guardar objetos, e que os outros componentes estão corretamente configurados.
3. Quando tudo estiver configurado e a aplicação instalada do lado do cliente, são definidos os parâmetros para comunicações e algoritmos de criptografia a usar entre a aplicação cliente

e o servidor da LOQR. A partir deste passo assume-se a disponibilidade do servidor da LOQR, assim como todas as comunicações existentes, incluindo também a disponibilidade do S3.

4. O cliente está a usar a aplicação e pretende adicionar um objecto novo à base de dados. Um pedido é feito ao servidor da LOQR contendo esse objeto, que será redirecionado para o SDK.
 5. O SDK gera uma nova chave e cifra o objecto com ela.
 6. O SDK acede ao Container para obter a chave mestra e conseguir assim cifrar a chave que foi usada para cifrar o objeto.
 7. O objeto cifrado e a chave cifrada são enviados para o S3, onde serão armazenados. Também são guardadas evidências de que o processo foi corretamente executado.
 8. O cliente está a usar a aplicação e pretende ler um objecto presente na base de dados. Um pedido é feito ao servidor da LOQR pedindo esse objeto.
 9. O Servidor da LOQR contacta a base de dados S3 e recebe o objeto pretendido, juntamente com a chave cifrada.
 10. O SDK acede ao Container para conseguir decifrar a chave cifrada, através da chave mestra.
 11. Após ter obtido a chave, o SDK prossegue em decifrar o objeto.
 12. O objeto é enviado e apresentado ao cliente.
- x. Durante os processos anteriores, o Operador Cliente poderá aceder a serviços de *back office*, sem qualquer ordem específica.
 - y. O Operador da LOQR em qualquer momento poderá aceder ao KeyManager e criar, editar ou eliminar chaves.

4.2.4 Ativos

Nesta parte do trabalho vai ser analisada a informação que os vários agentes valorizam neste sistema e quais as consequências da perda dessa informação.

- Cliente
 1. Este tendo adquirido o serviço, espera que os seus dados não sejam observados por nenhuma das outras partes existentes.
 2. Espera também que os dados sejam corretamente cifrados, podendo depois ser recuperada a versão original dos dados.
 3. Este valoriza a independência de auditorias externas que sejam feitas ao sistema, e que seja possível verificar qualquer anomalia nas várias componentes do sistema.

- Operador da LOQR
 1. A parte principal de segurança neste sistema é da responsabilidade deste agente, como tal, qualquer infração que ocorra pode denegrir a imagem deste agente.
 2. Como faz parte da empresa que oferece um serviço aos seus clientes, este valoriza que o serviço seja prestado por completo, herdando assim os ativos descritos para o agente anterior.
 3. O operador, fazendo parte da LOQR, valoriza as tecnologias que são usadas e as suas especificações, dado que podem representar um segredo de negócio.
- Administradores AWS, Microsoft SQL Server e ISP
 1. A percepção que os outros agentes têm da disponibilidade e fiabilidade dos seus servidores e as respectivas conexões, é o que estes agentes valorizam.

4.2.5 Modelo de Confiança

Nesta subsecção vão ser identificados certos comportamentos que são pressupostos:

1. Assume-se que o operador da LOQR ou outros funcionários, não divulgam qualquer tipo de informação confidencial.
2. Supõe-se que o Cliente não tem qualquer intenção de ser um atacante, tentar incriminar ou prejudicar qualquer outro agente do sistema, pois sendo o Cliente o único contacto direto com os seus clientes, em qualquer falha de segurança que possa existir, a responsabilidade será numa primeira fase atribuída ao Cliente, sendo assim a sua imagem que fica denegrida para os seus clientes.
3. Assume-se que os prestadores de serviços de bases de dados, irão respeitar o cumprimento das suas posições contratuais e assim:
 - (a) verificar que os processos de adicionar, pesquisar, alterar e apagar estão a funcionar corretamente,
 - (b) e que o serviço disponibiliza integridade, confidencialidade e disponibilidade para os seus clientes.
4. Existe algum mecanismo de redundância para assegurar a permanência dos dados nas bases de dados. Este mecanismo é unicamente do lado dos servidores de bases de dados.
5. As empresas detentoras dos serviços de bases de dados contêm os meios legais para apresentar processos em tribunais, caso algum administrador tenha algum comportamento indevido.
6. Assume-se que a LOQR já tenha implementado um sistema de prevenção e deteção de intrusões.

7. É pressuposto que auditorias externas feitas ao sistema sejam executadas por agentes honestos e independentes deste sistema.

4.3 Requisitos da Solução

Para a especificação dos requisitos será necessário presumir a existência de um componente de segurança que tenha capacidade de armazenar chaves criptográficas e geri-las de uma forma segura, no sentido que se definirá nesta secção. Este componente será referido como Componente Seguro (CS) durante esta secção, e nas seguintes será representado por um serviço específico. Este componente deverá ser criado e administrado pelo Operador Cliente, e este deverá dar o acesso externo necessário para as interações que se realizem. Como tal, a Figura 4.5 representa a arquitetura base, em que o CS está integrado.

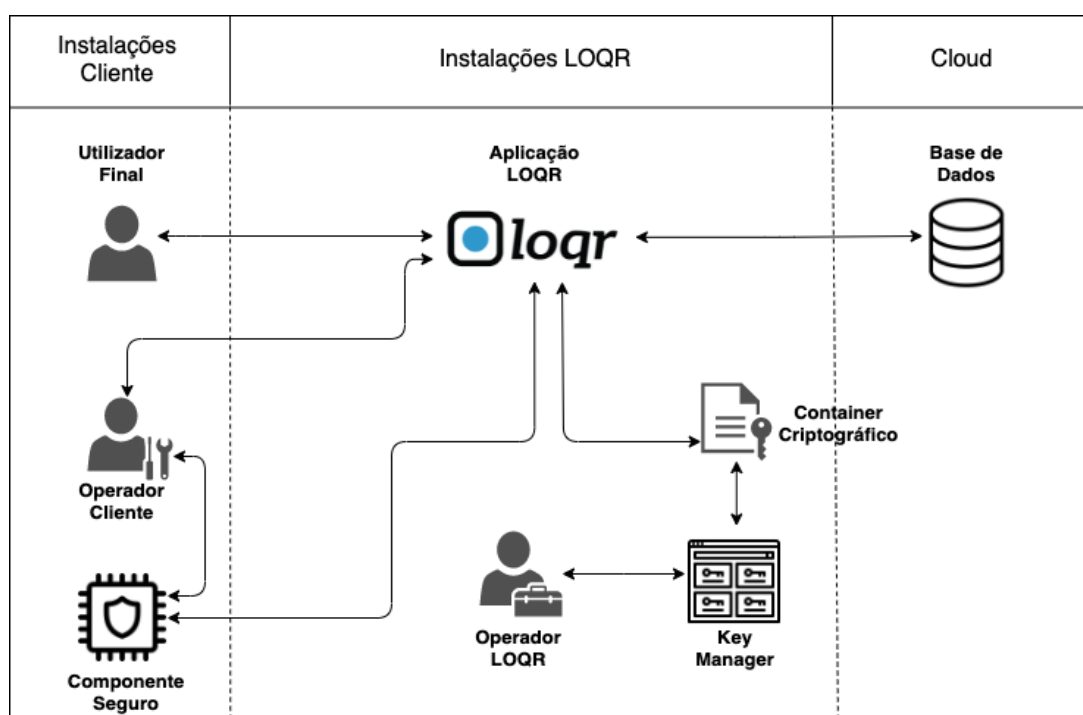


Figura 4.5: Arquitetura Base para cumprir requisitos.

Pode-se então distinguir os requisitos da Aplicação Global (referente ao sistema na sua totalidade), dos Requisitos do CS. Os requisitos da Aplicação Global são os seguintes:

- Os processos de cifração/decifração de dados e de chaves de cifra de dados deverão manter-se semelhantes ao original. No entanto, não existe a necessidade recorrer a *softwares* que estejam a ser usados atualmente.
- A Aplicação da LOQR deverá conseguir analisar os dados em trânsito, pois parte do serviço da LOQR passa por fazer um tratamento e organização de dados necessários no cumprimento de regras de negócio dos seus clientes. Apesar desse serviço ser fora do

contexto desta tese, é necessário ter em conta a Aplicação da LOQR deverá conseguir aceder a dados (em *plaintext*) no momento em que lhes são fornecidos, e antes de serem armazenados.

- A representação da base de dados deverá ser genérica, e, portanto, não ser específica a serviços da Microsoft ou da AWS. No entanto, há preferência de que o motor de base de dados seja um SQL Server.
- O único contacto que deverá existir entre a Aplicação da LOQR e o CS é para pedidos e respostas para cifrar e decifrar chaves de cifra de dados.
- A Aplicação LOQR deverá ser responsável por seleccionar a chave que pretende usar para um determinado processo, assim como de armazenar informação necessária para conseguir fazer pedidos de cifração/decifração a essa chave.
- Se o cliente retirar o acesso a chaves no CS ou o acesso ao CS, a Aplicação da LOQR não deverá conseguir ter acesso a nenhuma funcionalidade deste, e, portanto, o acesso a dados decifrados não será possível. No entanto, os dados cifrados deverão permanecer na base de dados para poderem ser acedidos quando o acesso ao CS voltar a ser disponibilizado.

Verifica-se a existência dos requisitos relativos ao Componente Seguro:

- Os clientes deverão conseguir aceder a este componente, e através de um processo fácil, alterar as chaves mestras que lhe estão associadas.
- O acesso de clientes a este componente deverá ser autenticado e restringido.
- Clientes que sejam autenticados não deverão ter acesso a chaves mestras de outros clientes.
- A Aplicação LOQR deverá conseguir comunicar com este componente.
- Chaves mestras criptográficas neste componente permanecem durante todo o seu período de vida apenas no seu ambiente, e, portanto, nunca deverão estar expostas à Aplicação LOQR ou a qualquer outro elemento. Como tal, não deverá existir a possibilidade de as chaves serem exportadas.
- Este componente deverá ceder acesso à Aplicação LOQR, mas unicamente para as chaves mestras e respetivas funções necessárias.
- O Operador Cliente, que até a este ponto seria apenas responsável por aceder a serviços de *back-office*, será o responsável pela criação, gestão e deste componente, assim como de fornecer as informações necessárias à LOQR para que o acesso seja possível.

Agora que são perceptíveis os requisitos necessários para o sistema pretendido, será possível propor no próximo capítulo, uma solução que tenha como objetivo alcançá-los.

Capítulo 5

Concepção

Numa primeira fase são explicadas na secção 5.1 possíveis soluções, seguidas na secção 5.2 de uma proposta de alteração do sistema de gestão de chaves para serviços em que a chave é gerida pelo utilizador, e na secção 5.3 finaliza-se com a justificação de que essa proposta cumpre os requisitos apresentados no capítulo anterior.

Nas secções 5.2 e 5.3, será possível verificar que os requisitos principais para a alteração do sistema são alcançáveis, adicionando a possibilidade de um utilizador usar uma chave criptográfica que esteja em sua posse, minimizando as alterações que são feitas ao serviço fornecido pela LOQR, e mantendo parte do sistema atual de gestão de chaves, de modo a possibilitar uma transição fácil para utilizadores não considerem necessário esse controlo de chaves.

Parte das decisões em relação a que tecnologias usar têm em conta os interesses da LOQR, e, portanto, foram tomadas em conjunto.

5.1 Racional

Numa primeira observação de como proporcionar um armazenamento seguro em que o controlo da chave pertença ao fornecedor dos dados, a resposta simples seria que o utilizador gerasse a sua própria chave e cifrasse a sua informação localmente, sendo que a Aplicação LOQR, quando tivesse acesso aos dados, estes já estavam cifrados. Para além de a computação de cifrar/decifrar ter de ser integrada no computador de cada utilizador, o que leva a gastos energéticos superiores do lado do utilizador, a chave seria armazenada localmente no computador de cada utilizador, o que reduz o nível de segurança.

Uma solução para aumentar esse nível de segurança seria fazer um pré-processamento, num Componente Seguro, de dados que fossem assinalados como confidenciais. Desse modo, a chave seria armazenada num componente seguro, o que facilitaria a gestão de chaves criptográficas. No entanto tal solução não é possível, pois não cumpre um dos requisitos descritos anteriormente, em que é referido que um dos serviços da LOQR necessita de aceder a dados em *plaintext* antes

de serem cifrados e armazenados.

É assim necessário que a solução passe por cifrar os dados depois de estes serem recebidos pela Aplicação LOQR, mantendo o fluxo de dados semelhante ao original, apenas alterando a localização e acessos às chaves criptográficas.

Outra solução seria dar a possibilidade ao cliente, de num determinado momento, gerar na aplicação uma chave mestra que seria guardada localmente, e que também seria enviada à Aplicação da LOQR. Com esta a chave mestra, a Aplicação da LOQR re-cifrava as chaves previamente cifradas, **mas não armazenando a chave mestra**, impedindo assim posteriores acessos às chaves usadas para cifrar dados, e consequentemente, também a dados cifrados com essas chaves. Para os dados voltarem a estar acessíveis, o cliente teria de fornecer a chave mestra à LOQR, para que esta pudesse decifrar as chaves.

Esta solução parte do princípio que a Aplicação da LOQR é honesta neste processo de re-cifra, e que portanto, realmente re-cifra as chaves e que não armazena a chave mestra. Ficou então decidido que esta seria uma opção que não deveria ser desenvolvida, pois o processo de restrição de decifra de dados deve ser independente das componentes controladas pela LOQR, e que se essa funcionalidade vai ser desenvolvida, então o cliente deverá ter um maior controlo da chave mestra do que esta solução propõe.

Como tal, a solução proposta passará por usar tecnologias de gestão de chaves alterando apenas o local onde a chave é armazenada e a cifração/decifração é feita. Estas tecnologias serão debatidas nas próximas secções.

5.2 Solução Proposta

Será então proposto um sistema, com tecnologias adicionais, que visam cumprir os requisitos pretendidos na secção 4.3.

São apresentadas duas tecnologias de gestão de chaves, o Key Vault da Azure e o KMS da AWS, que vão de acordo com os interesses industriais da LOQR. É demonstrada a possibilidade de integrar essas tecnologias adicionais no sistema, com uma ligeira alteração, para alcançar as novas funcionalidades pretendidas. Estas mudanças também não tornam inutilizável o estado atual, pois foi proposto um sistema que para clientes que não pretendam tais requisitos, podem usar o sistema, mas sem terem controlo da sua chave mestra, de forma semelhante ao mostrado serviço atual.

Serão então descritas as Decisões Tecnológicas tomadas, a Arquitetura do sistema e os seus componentes, e o Fluxo de Dados e possíveis ações.

5.2.1 Decisões Tecnológicas

Nesta secção serão explicadas as decisões que foram tomadas em relação a que tecnologias usar para representar certas partes do sistema. Também será explicado o porquê de se optar por não integrar determinadas tecnologias que já estão implementadas no serviço da LOQR, optando assim por criar versões alternativas e simplificadas das mesmas.

5.2.1.1 Alternativa de Cifra de Dados

Dado o Always Encrypted ser uma tecnologia específica para serviços da Microsoft e o SDK do S3 também exclusivo para produtos da Amazon, pretendeu-se que o sistema fosse independente do *provider* de serviços de base de dados e que funcionasse de forma genérica para todas as bases de dados.

Como tal, as cifras usadas para cifrar dados são fornecidas pelo Java, dado que é possível utilizar algoritmos baseados em AES, algoritmos estes que o Always Encrypted e o SDK do S3 usam, e assim simular o comportamento destas ferramentas, mas com a generalidade referida em cima. Dado que nenhuns dos dados serão indexados, os IVs serão gerados aleatoriamente, resultando em *ciphertexts* diferentes para *inputs* iguais.

Para utilizadores que possuam serviços de gestão de chaves criptográficas, a cifra de chaves simétricas deixará de ser feita na Aplicação Web, e passará a ser feita nesses serviços. Para utilizadores que não possuam esses serviços e que escolham continuar a usar o estado atual do sistema, a cifra de chaves simétricas recorrendo a algoritmos baseados em RSA, permanecerá semelhante ao serviço da LOQR, mantendo a chave no controlo da LOQR.

5.2.1.2 Base de Dados e Tabelas

Aqui também se pretendia que o sistema fosse independente do serviço de *cloud* onde a base de dados fosse hospedada. No entanto, houve preferência que o motor de base de dados fosse um SQL Server. Como tal, foi hospedado um SQL server na Azure e outro na AWS, para demonstrar a possibilidade do sistema funcionar em diferentes fornecedores de serviços.

Para além de ser pretendida independência do *provider*, também se pretendia que as tabelas na base de dados fossem representáveis em diferentes motores de bases de dados. Portanto, foram escolhidos tipos de dados e *keywords* que são genéricas a vários motores de bases de dados ou que possibilitem transicionar para alternativas semelhantes.

Pode-se ver o esquema UML na Figura 5.1 composta pelas quatro tabelas que foram criadas para suportar o armazenamento necessário:

- Clients - Esta tabela representa um utilizador da aplicação e contém um ID, um *username* e uma *password*. Para além disso, contém um *role* e uma opção *enable*. Estes dois últimos

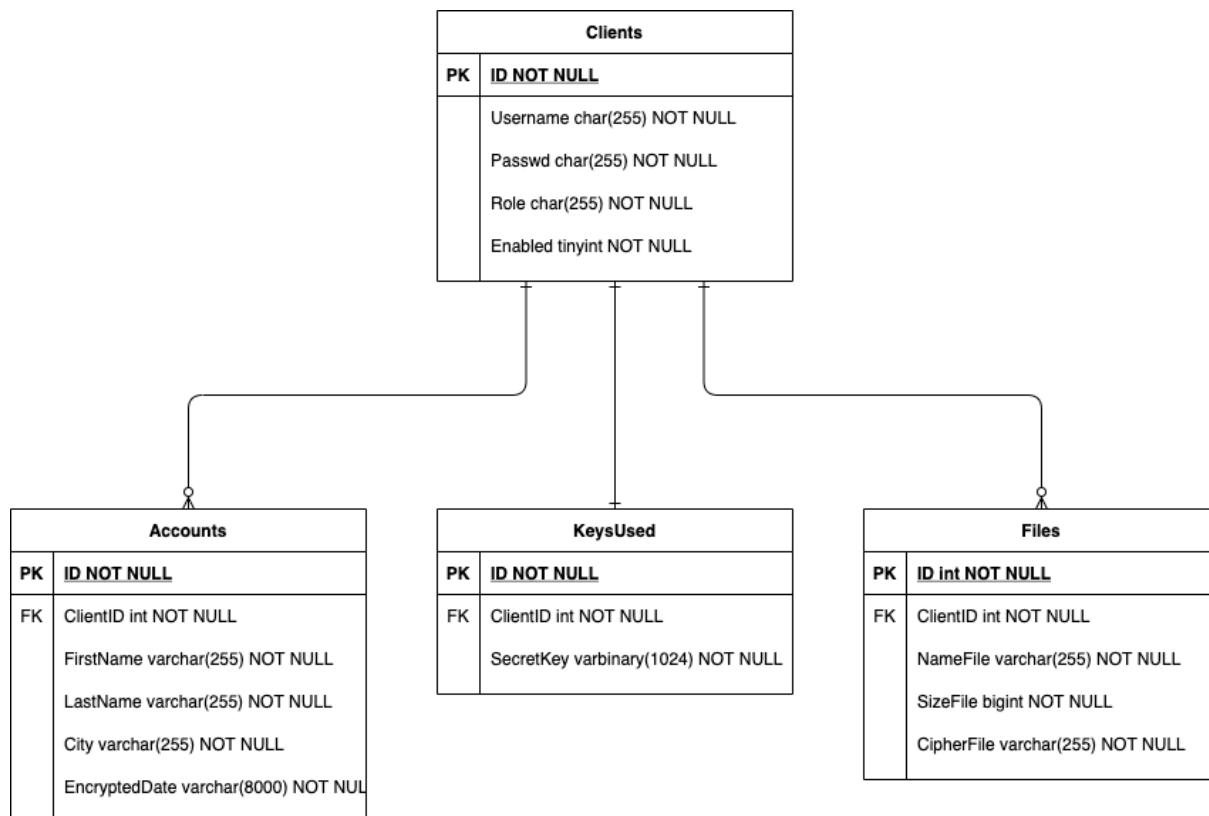


Figura 5.1: UML esquema da base de dados.

dados foram apenas criados porque o Spring já contém uma *framework* que permite fazer um *login*, que necessita desses dados.

- **Accounts** - Nesta tabela serão armazenadas informações de contas pertencentes a cada utilizador. É então armazenado um ID, o ID do cliente a que se refere, três *strings* que apenas foram estão presentes para a representação ser mais realista, e um valor binário *EncryptedData* que representa os dados que serão guardados cifrados. Esta tabela representará a parte do serviço associado ao Always Encrypted.

Usou-se o tipo de dados *VARBINARY*, que pode receber como argumento o tamanho, podendo este ser até um limite de 8000 *bytes* ou especificando com o valor *MAX*, passando assim a ter um tamanho de 2 GB. Várias bases de dados suportam esta variável ou alguma semelhante binária, podendo assim este esquema da tabela ser adaptado a outras bases de dados. No protótipo usou-se apenas 1024 *bytes*, mas poderão ser usados valores maiores, e mesmo recorrer a *BLOBs* (Binary Large Object).

- **Files** - Aqui ficarão armazenados ficheiros de que o cliente faça *upload* e posteriormente *download*, e pretende simular os serviços fornecidos pela LOQR, que têm integrado o armazenamento S3. Assim como a outras tabelas, esta tabela contém um ID e um ID do cliente a quem o ficheiro pertence, mas também o nome e o tamanho de ficheiros.

Adicionalmente é armazenado o ficheiro cifrado usando o tipo de dados *VARBINARY*, como na tabela anterior, mas agora com um tamanho de 8000 *bytes*. Como foi referido no

ponto anterior, se for pretendido um tamanho superior, poderá ser usado com o argumento MAX ou também poderão ser usados BLOBs através da *keyword* FILESTREAM.

- KeysUsed - Aqui apenas serão guardadas as chaves simétricas cifradas num VARBINARY, e, mais uma vez, um ID e o ID do cliente a quem esta pertence.

5.2.2 Arquitetura

Na Figura 5.2 pode-se se ver um esquema do sistema proposto, que é semelhante ao do sistema atual da LOQR, mas com o Key Vault e o KMS. Existem os seguintes componentes:

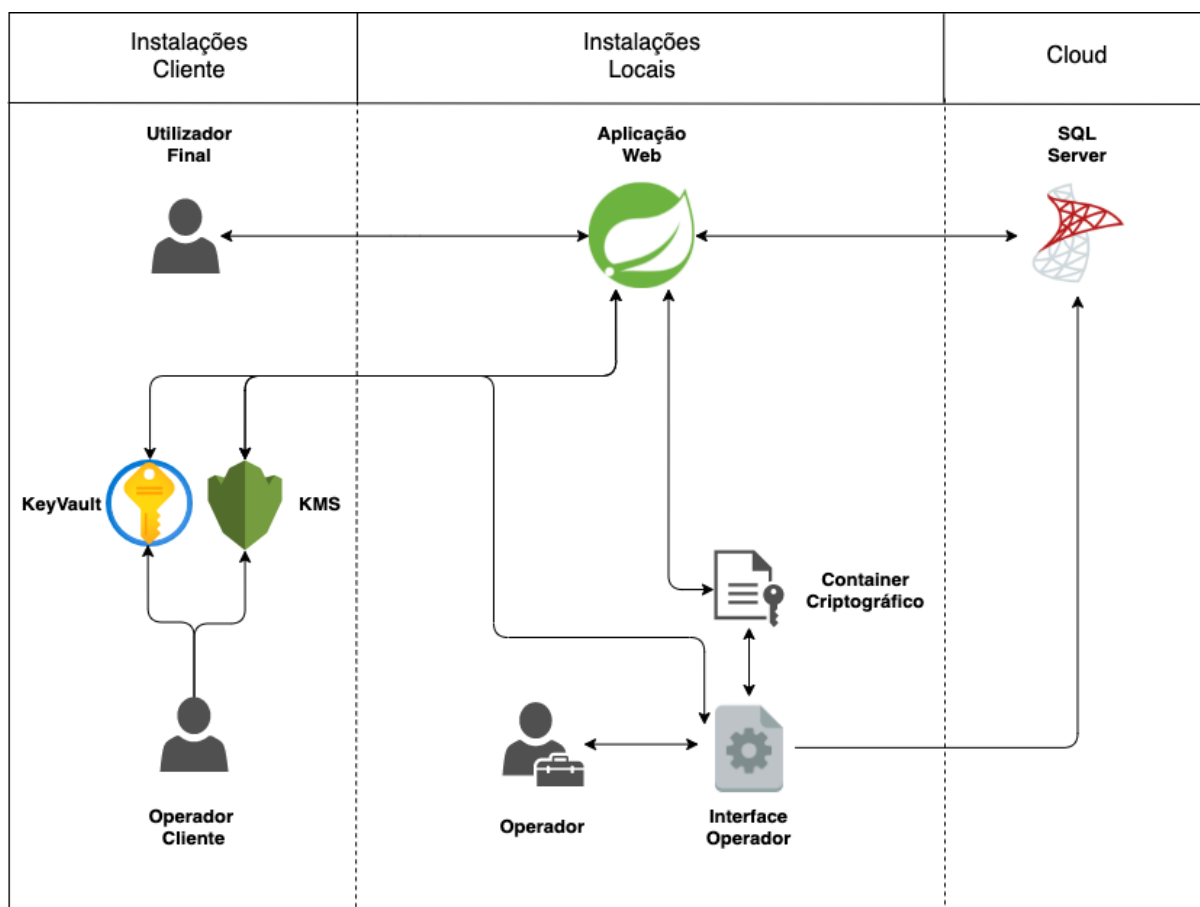


Figura 5.2: Arquitetura do Sistema Proposto.

- Utilizador Final - Este componente representa o acesso feito através do *browser* dos clientes, não sendo necessária qualquer gestão por parte dos clientes. Este contém várias funcionalidades, mas nenhuma relacionada com segurança, sem ser a de comunicar os dados à Aplicação Web. É através deste componente que os clientes interagem com o sistema. É de notar que num caso real, os pedidos web deverão ser protegidos através de TLS.
- Aplicação Web - Este servidor opera como intermediário entre o cliente e a base de dados SQL. A sua função é cifrar partes das *queries* do cliente e transmiti-las para o SQL Server e

decifrar a resposta obtida relativamente à *query*, e transmiti-la ao cliente. Esta cifra recorre a uma decifra da chave simétrica cifrada usando a chave mestra. Se for um utilizador que tenha acesso a um serviço de gestão de chaves (Key Vault ou KMS), a decifra dessa chave simétrica cifrada é feita nesses componentes, caso contrário, será neste componente, de forma semelhante ao serviço atual da LOQR.

- Operador - Este apenas representa o operador que interage com a Interface Operador.
- Interface Operador - Neste componente da arquitetura o Operador pode criar, alterar e eliminar chaves simétricas que serão usadas pela Aplicação Web para utilizadores que não tenham um serviço de gestão de chaves. Para a criação de chaves é usado o Java keytool. No entanto, essas chaves durante o seu ciclo de vida irão ficar guardadas no Container. Também pode criar contas para novos utilizadores, e adicionar credenciais de acesso a chaves presentes em serviços de gestão de chaves pertencentes a um utilizador.
- Container Criptográfico - Este componente é um ficheiro P12 que armazena chaves criptográficas. Este ficheiro é protegido através de uma *password*. Aqui são armazenadas as chaves mestras dos clientes que não têm um serviço de gestão de chaves.
- Operador Cliente - Este deverá criar chaves criptográficas no serviço de gestão de chaves pertencente ao cliente (KeyVault ou Azure) e de fornecer as credencias de acesso a essas chaves ao Operador. Também tem a capacidade de gerir essas chaves, e de nomeadamente, fazer *enable* e *disable* destas.
- Key Vault - Este é o serviço de gestão de chaves da Azure, pertencente ao cliente. Chaves criadas neste serviço, nunca serão exportadas (excluindo chaves públicas de um par de chaves assimétricas).
- KMS - Este é o serviço de gestão de chaves da AWS, pertencente ao cliente. Chaves criadas neste serviço, nunca serão exportadas (excluindo chaves públicas de um par de chaves assimétricas).
- SQL Server - Esta base de dados é criada e administrada pela por Operadores. Esta recebe os dados enviados pela Aplicação Web e armazena-os.

5.2.2.1 Azure Key Vault

No portal da Azure é possível criar vários Key Vaults, e em cada Key Vault criar pares de chaves assimétricas, RSA e EC. Registrando aplicações através do Active Directory, será possível ceder o acesso a uma determinada Aplicação ao nível do Key Vault. Ou seja, nesta ferramenta todos os utilizadores que tiverem o acesso a um determinado Key Vault, têm acesso a todas as chaves neste, não sendo possível distinguir o acesso dentro do Key Vault. No entanto, permite facilmente atribuir diferentes permissões dentro do Key Vault, como será demonstrado no próximo capítulo, assim como outras funcionalidades desta ferramenta.

5.2.2.2 AWS KMS

Na consola da AWS pode-se criar chaves independentes, sendo esta ferramenta, nesse aspeto, bastante diferente da anterior. Aqui o controlo de chaves é feito ao nível das chaves, podendo-se dar o acesso a uma chave a diferentes utilizadores. No entanto não tem um leque de permissões que se pode dar a cada utilizador. Podem ser criadas chaves simétricas ou assimétricas.

No próximo capítulo, será mostrado como criar uma chave, permitir o acesso, e os comportamentos possíveis.

5.2.2.3 P12

Apesar do foco desta solução ser passar o controlo da chave para o cliente, mantém-se a possibilidade do cliente optar por não necessitar de tal funcionalidade. Como tal, pode-se manter um sistema de chaves igual ao do serviço atual da LOQR. Assim, também será possível manter alguma compatibilidade com o sistema anterior, e permitir uma transição mais fácil.

5.2.3 Fluxos de Dados e Ações

O Fluxo de Dados desta solução é de certa forma semelhante ao do serviço da LOQR, dado que a alteração está no local onde se computa a cifração/decifração das chaves AES. Nas seguintes ações e fluxos de dados assume-se que o acesso e permissões a chaves criptográficas presentes em serviços de gestão de chaves foram corretamente cedidos.

Pode-se ver os seguintes fluxos de dados e ações presentes na Figura 5.3 e a legenda nos seguintes pontos (apenas os dois primeiros pontos devem ser nessa ordem, podendo os outros ser em ordens diferentes, desde que sejam depois dos dois primeiros; os sub-pontos respeitam a ordem dentro de cada ponto):

- Inicialmente será necessário configurar o sistema para aceitar o utilizador:
 1. O Utilizador Final contacta o Operador, para que consiga obter credenciais de acesso. Se pretender usar um serviço de gestão de chaves que possua, deverá fornecer as credencias de acesso à chave.
 2. O Operador cria as credenciais para o Utilizador conseguir proceder com o *login* e fornece-as por um outro canal (por exemplo, pessoalmente).
 3. É gerada uma entrada na *keystore* para uma chave mestra pertencente ao cliente, caso este não tenha um serviço de gestão de chaves. Também é gerada uma chave AES, que depois é cifrada com a chave presente na *keystore*, se o cliente não tiver serviço de gestão de chaves, ou cifrada no Key Vault ou no KMS, de acordo com o serviço de gestão de chaves do utilizador. Esta chave cifrada é armazenada na base de dados.
- O Utilizador realiza o processo de autenticação necessário:

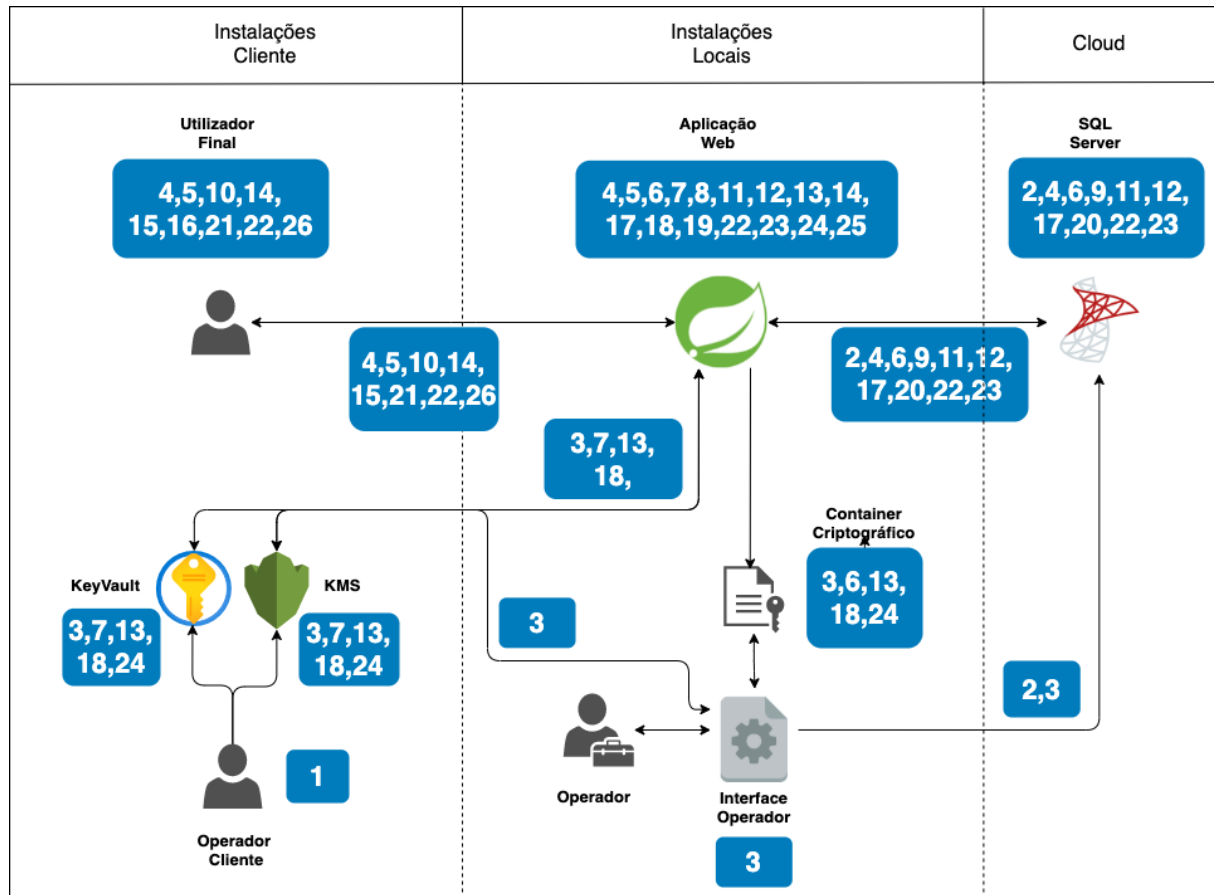


Figura 5.3: Ações e Fluxos de Dados do Sistema Proposto.

4. O Utilizador faz o processo de *login*.
- Adiciona uma conta nova:
 5. O Utilizador faz um pedido para adicionar uma conta nova, contendo todos os dados necessários.
 6. A Aplicação Web seleciona qual o serviço de gestão de chaves pertencente ao cliente e qual a chave AES cifrada presente na base de dados.
 7. Se o cliente não tiver um serviço de gestão de chaves, a sua chave mestra é selecionada através do ficheiro P12 e decifra-se a chave AES cifrada, obtendo a sua versão *plaintext*. Se o cliente tiver um serviço de gestão de chaves, é feito um pedido ao serviço para decifrar a chave AES cifrada.
 8. Com a chave AES, a Aplicação Web cifra os dados correspondentes à coluna EncryptedData.
 9. O conjunto de dados cifrados e não cifrados (colunas para além da EncryptedData) são adicionados à base de dados.
 - É pretendido o acesso às informações de uma conta:
 10. O Utilizador faz um pedido para visualizar as suas contas associadas.

11. É feita uma pesquisa para obter os dados pretendidos.
 12. A Aplicação Web seleciona qual o serviço de gestão de chaves pertencente ao cliente e qual a chave AES cifrada presente na base de dados.
 13. Se o cliente não tiver um serviço de gestão de chaves, a sua chave mestra é selecionada através do ficheiro P12 e decifra-se a chave AES cifrada, obtendo a sua versão *plaintext*. Se o cliente tiver um serviço de gestão de chaves, é feito um pedido ao serviço para decifrar a chave AES cifrada.
 14. Com a chave AES, a Aplicação Web decifra os dados correspondentes à coluna EncryptedData.
 15. Os dados são enviados em resposta ao Utilizador Final.
- O utilizador pretende armazenar um ficheiro novo.
 16. O Utilizador faz *upload* de um ficheiro.
 17. A Aplicação Web seleciona qual o serviço de gestão de chaves pertencente ao cliente e qual a chave AES cifrada presente na base de dados.
 18. Se o cliente não tiver um serviço de gestão de chaves, a sua chave mestra é selecionada através do ficheiro P12 e decifra-se a chave AES cifrada, obtendo a sua versão *plaintext*. Se o cliente tiver um serviço de gestão de chaves, é feito um pedido ao serviço para decifrar a chave AES cifrada.
 19. Com a chave AES cifra o ficheiro.
 20. O ficheiro cifrado é armazenado na base de dados.
 - O utilizador pretende aceder a um ficheiro.
 21. O Utilizador pretende fazer *download* de um dos seus ficheiros e envia um pedido requisitando informações de todos os seus ficheiros.
 22. A Aplicação Web seleciona os nomes e tamanhos (armazenado em *plaintext*) de ficheiros e devolve ao Utilizador. O Utilizador depois seleciona o ficheiro pretendido.
 23. A Aplicação Web seleciona qual o serviço de gestão de chaves pertencente ao cliente e qual a chave AES cifrada presente na base de dados.
 24. Se o cliente não tiver um serviço de gestão de chaves, a sua chave mestra é selecionada através do ficheiro P12 e decifra-se a chave AES cifrada, obtendo a sua versão *plaintext*. Se o cliente tiver um serviço de gestão de chaves, é feito um pedido ao serviço para decifrar a chave AES cifrada.
 25. Com a chave AES, a Aplicação Web decifra o ficheiro.
 26. O ficheiro decifrado é enviado ao Utilizador Final.

5.3 Cumprimento dos Requisitos

Na secção 4.3 foram especificados quais os requisitos que são pretendidos para a solução pretendida. Parte desses requisitos serão relativos à totalidade do sistema, e outros apenas aos componentes responsáveis pela gestão de chaves criptográficas.

Assim sendo, pode ser feita uma correspondência entre os requisitos pretendidos e a justificação do cumprimento dos mesmos. Nas Tabelas 5.1 e 5.2 podem ser vistas estas correspondências, relativas ao requisitos totais do sistema e dos componentes de gestão de chaves criptográficas, respetivamente.

Tabela 5.1: Cumprimento dos requisitos do Sistema Global

Requisitos	Justificação do Cumprimento
A cifração/decifração de dados e de chaves de cifra de dados deverão manter-se semelhantes ao original. No entanto, não existe a necessidade recorrer a <i>softwares</i> que estejam a ser usados atualmente.	A cifração/decifração de dados recorrendo a chaves simétricas é semelhante às cifras usadas pelos <i>softwares</i> utilizados no serviço atual da LOQR. Tais cifras são executadas na Aplicação Web. A cifração/decifração de chaves simétricas, recorrendo a cifras assimétricas, será transparente para a Aplicação Web, que apenas terá de fazer pedidos de cifração/decifração ao serviço de gestão de chaves criptográficas (caso o acesso às chaves em questão seja permitido).
A Aplicação da LOQR deverá conseguir analisar os dados em trânsito, pois parte do serviço da LOQR passa por fazer um tratamento e organização de dados necessários no cumprimento de regras de negócio dos seus clientes. Apesar desse serviço ser fora do contexto desta tese, é necessário ter em conta a Aplicação da LOQR deverá conseguir aceder a dados (em <i>plaintext</i>) no momento em que lhes são fornecidos, e antes de serem armazenados.	A Aplicação Web desenvolvida tem acesso a dados em <i>plaintext</i> antes de os cifrar.
A representação da base de dados deverá ser genérica, e, portanto, não ser específica a serviços da Microsoft ou da AWS. No entanto, há preferência de que o motor de base de dados seja um SQL Server.	Foram escolhidos tipos de dados no esquema de base de dados que são comuns a vários motores de bases de dados. Na arquitetura a base de dados é representada por um SQL Server.
O único contacto que deverá existir entre a Aplicação da LOQR e o CS é para pedidos e respostas para cifrar e decifrar chaves de cifra de dados.	O acesso a chaves presentes no serviço de gestão de chaves é feito através das credenciais fornecidas pelo Operador Cliente. O Operador Cliente também tem a capacidade de restringir as operações associadas às chaves.
A Aplicação LOQR deverá ser responsável por selecionar a chave que pretende usar para um determinado processo, assim como de armazenar informação necessária para conseguir fazer pedidos de cifração/decifração a essa chave.	Quando existe a necessidade de aceder a uma chave, a Aplicação Web seleciona a chave e as credenciais associadas ao cliente em questão.
Se o cliente retirar o acesso a chaves no CS ou o acesso ao CS, a Aplicação da LOQR não deverá conseguir ter acesso a nenhuma funcionalidade deste, e portanto, o acesso a dados decifrados não será possível. No entanto, os dados cifrados deverão permanecer na base de dados para poderem ser acedidos quando o acesso ao CS voltar a ser disponibilizado.	O Operador Cliente através do portal da Azure ou da consola da AWS consegue fazer <i>enable/disable</i> a chaves mestras que lhe pertençam. Se uma chave mestra for <i>disabled</i> , as chaves simétricas presentes na base de dados não serão posteriormente decifráveis, e consequentemente, os dados cifrados com estas. Se o Operador Cliente futuramente fizer <i>enable</i> da chaves mestra, a Aplicação Web, voltará a conseguir aceder a dados.

Tabela 5.2: Cumprimento dos requisitos do Serviço de Gestão de Chaves Criptográficas

Requisitos	Justificação do Cumprimento
Os clientes deverão conseguir aceder a este componente, e através de um processo fácil, alterar as chaves mestras que lhe estão associadas.	O cliente será o responsável por criar este componente, e consequentemente, terá acesso ao portal da Azure ou à consola da AWS, onde pode realizar operações associadas aos seus serviços de gestão de chaves.
O acesso de clientes a este componente deverá ser autenticado e restringido.	O acesso aos serviços de gestão de chaves serão associados às plataformas do cliente na AWS ou na Azure, que já possuem métodos de autenticação.
Clientes que sejam autenticados não deverão ter acesso a chaves mestras de outros clientes.	Não existe partilha de serviços de gestão de chaves ou credenciais entre clientes.
A Aplicação LOQR deverá conseguir comunicar com este componente.	O acesso a estes serviços através da Aplicação Web é permitido e configurável pelo Operador Cliente, através da plataforma do serviço de gestão de chaves.
Chaves mestras criptográficas neste componente permanecem durante todo o seu período de vida apenas no seu ambiente, e portanto, nunca deverão estar expostas à Aplicação LOQR ou qualquer outro elemento. Como tal, não deverá existir a possibilidade de as chaves serem exportadas.	Estes serviços de gestão de chaves criptográficas não permitem que chaves simétricas ou chaves privadas pertencentes a pares assimétricos sejam exportadas. No entanto, chaves públicas pertencentes a pares de chaves assimétricas podem ser exportadas, dado que não permitem a decifra. Neste contexto, isso não é uma limitação porque apenas permite que novas chaves simétricas sejam cifradas e adicionadas ao sistema, não podendo depois ser decifradas, continuando assim o acesso a dados restrito.
Este componente deverá ceder acesso à Aplicação LOQR, mas unicamente para as chaves mestras e respetivas funções necessárias.	O Operador Cliente tem a capacidade de, através da plataforma do seu serviço de gestão de chaves criptográficas, restringir as operações e permissões dos utilizadores que têm acesso às chaves criadas por este.
O Operador Cliente, que até a este ponto seria apenas responsável por aceder a serviços de <i>back-office</i> , será o responsável pela criação, gestão e deste componente, assim como de fornecer as informações necessárias à LOQR para que o acesso seja possível.	O Operador Cliente cria o serviço de gestão de chaves na plataforma pretendida, e por um canal externo (por exemplo, presencialmente) fornece as credenciais de acesso à LOQR.

Capítulo 6

Implementação e Resultados Experimentais

Na secção 6.1 deste capítulo será descrito o desenvolvimento de um protótipo que numa fase inicial represente o estado atual do serviço fornecido pela LOQR, sendo que no capítulo 4 foram entendidas todas as componentes relevantes para representar esse estado atual. Posteriormente, na secção 6.2 será possível progredir e alterar o protótipo para que se consiga alcançar os requisitos apresentados na secção 4.3, segundo a proposta feita na secção 5.2.

Neste capítulo serão apresentados alguns detalhes mais técnicos relativamente às tecnologias aplicadas, nomeadamente dos serviços de gestão de chaves criptográficas.

6.1 Primeira Fase - Estado Atual

No início do capítulo 4 começou-se por detalhar o estado atual do serviço fornecido pela LOQR. Para que posteriormente seja possível implementar as funcionalidades pretendidas, será necessário que o protótipo numa fase inicial represente as tecnologias usadas atualmente. Algumas já foram ligeiramente debatidas no capítulo anterior, mas nesta secção serão detalhadas algumas mais específicas à primeira fase.

6.1.1 Arquitetura

Nesta subsecção serão descritos os componentes da Arquitetura Base do sistema desenvolvido nesta fase (presente na Figura 6.1).

A Instalações Cliente representam onde o cliente interage com o sistema, que no caso deste protótipo será localmente através de um *browser*, as Instalações Locais representam o espaço onde a Aplicação Spring e as chaves criptográficas serão armazenadas, que nesta primeira fase também será localmente, e a Cloud representa uma plataforma *cloud* das que foram usadas neste

protótipo. Recordo que na implementação deste protótipo o SQL Server foi implementado em serviços da Azure e da AWS.

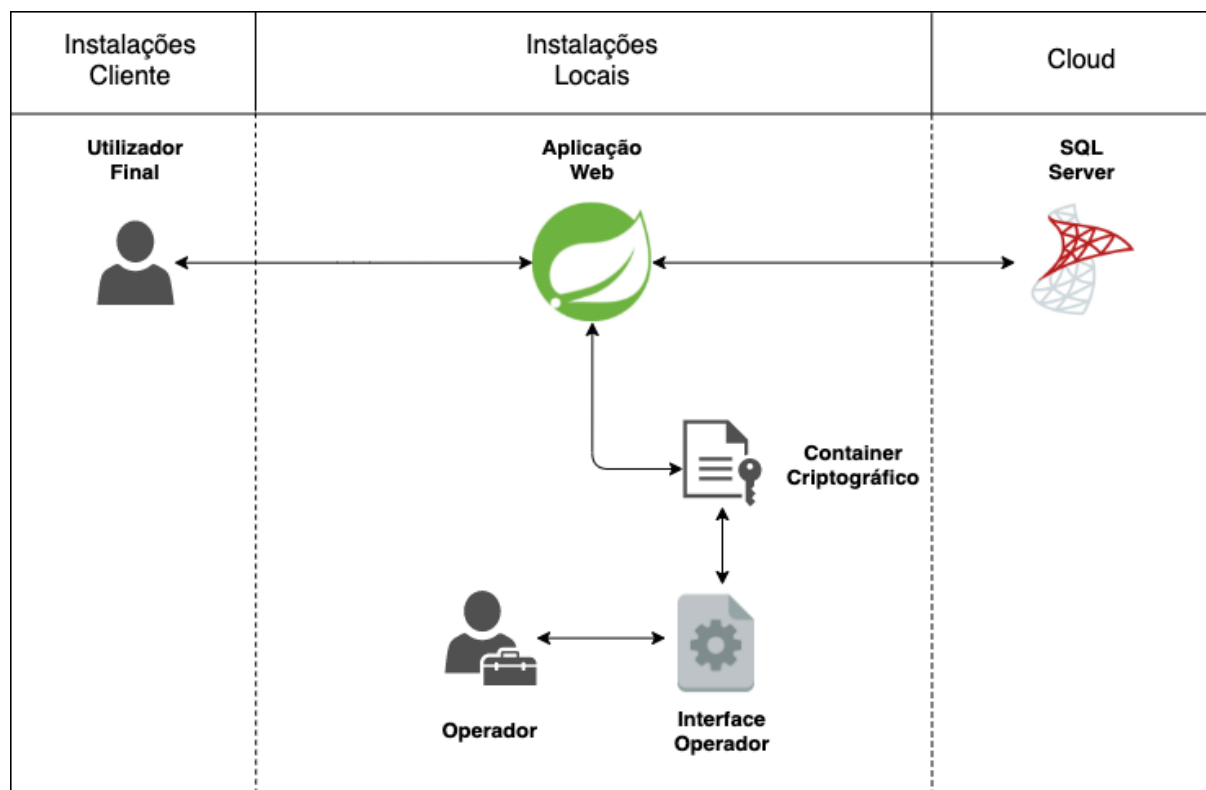


Figura 6.1: Arquitetura da Primeira Fase do Protótipo.

Como se tentou manter a arquitetura deste protótipo o mais semelhante possível à arquitetura do serviço da LOQR, muito dos componentes desta acabam por ser semelhantes, mas com ligeiras alterações. Dado que nesta fase o Operador Cliente não tem qualquer função, foi excluído da Arquitetura. Podemos então verificar a existência dos seguintes componentes:

- **Utilizador Final** - Este componente representa o acesso feito através do *browser* dos clientes, não sendo necessária qualquer gestão por parte dos clientes. Este contém várias funcionalidades, mas nenhuma relacionada com segurança, sem ser a de comunicar os dados à Aplicação Web. É através deste componente que os clientes interagem com o sistema. É de notar que num caso real os pedidos *web* deverão ser protegidos através de TLS.
- **Aplicação Web** - Este servidor opera como intermediário entre o cliente e a base de dados SQL. A sua função é cifrar partes das *queries* do cliente e transmiti-las para o SQL Server e decifrar a resposta obtida relativamente à *query*, e transmiti-la ao cliente.
- **Operador** - Este apenas representa o operador que interage com a Interface Operador.
- **Interface Operador** - Neste componente da arquitetura o Operador pode criar, alterar e eliminar chaves simétricas que serão usadas pela Aplicação Web. Para a criação de chaves é

usado o Java keytool. No entanto, as chaves durante o seu ciclo de vida irão ficar guardadas no Container. Também pode criar contas para novos utilizadores.

- Container Criptográfico - Este componente é um ficheiro P12 que armazena chaves criptográficas. Este ficheiro é protegido através de uma *password*. Aqui são armazenadas as chaves mestras de cada cliente.
- SQL Server - Esta base de dados é criada e administrada por Operadores. Esta recebe os dados enviados pela Aplicação Web e armazena-os.

6.1.1.1 Aplicação Web e Spring

A escolha do *software* utilizado para simular a Aplicação LOQR acaba por não ser um ponto muito relevante para esta tese, tendo sido a escolha deixada em aberto, desde de que represente as funcionalidades pretendidas. A Aplicação LOQR foi desenvolvida em Spring, o que acabou por facilitar a escolha, pois aqui seria um ponto fácil de representar o serviço atual, e não existia nenhuma limitação para não fazer tal. Como tal, acabou por se escolher desenvolver a Aplicação Web em Java, dado que possibilita utilizar as ferramentas descritas, e assim manter mais semelhança com o sistema original.

Para facilitar o processo de *login* foi usada a *Framework Spring Security*, que automaticamente cria uma página de *login* e permite facilitar o processo de autenticação. Neste processo apenas terá se ser fornecida um *query* que seleciona o utilizador, e selecionar também como a *password* foi codificada. O Spring Security recorre ao **bcrypt**, um função de *hash* baseada em *Blowfish*. A resultante *hash* é do seguinte formato : "\$2a\$10\$sZvsve/.rc8Euml1rZwp4.kfDGPTXqRwVReqlh.qI5EOoKtBGGRK", em que "\$2a" representa o algoritmo usado (bcrypt), "\$10" um parâmetro para o número de rondas, e dos restantes caracteres Radix-64, 128 *bits* representam o *salt*, e os restantes 192 representam a *hash*.

Naturalmente isto requer que na criação de um novo utilizador, que ocorre no processo de registo feito através de um painel acessível ao administrador (Figura 6.2), que a *hash* da *password* seja armazenada dessa mesma forma.

Foram então criados *mappings* para dar resposta aos vários tipos de pedidos que poderão ser feitos por um cliente. Nestes *mappings* é onde ocorre a maior parte do processamento, acessos a chaves, cifras/decifras e acessos à base de dados. Podem-se ver nas Figuras 6.3, 6.4, 6.5, 6.6, 6.7 e 6.8 páginas da Aplicação Web com as quais o utilizador interage.

Neste serviço existem as seguintes páginas em que o Utilizador Final interage:

- Login (Figura 6.3).

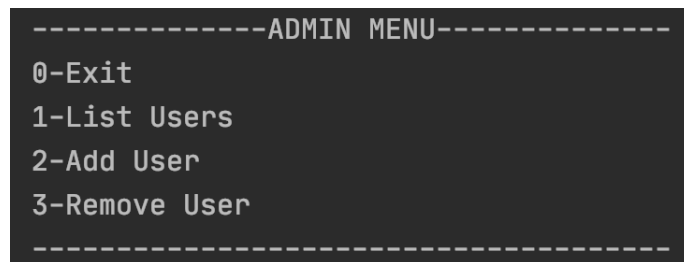
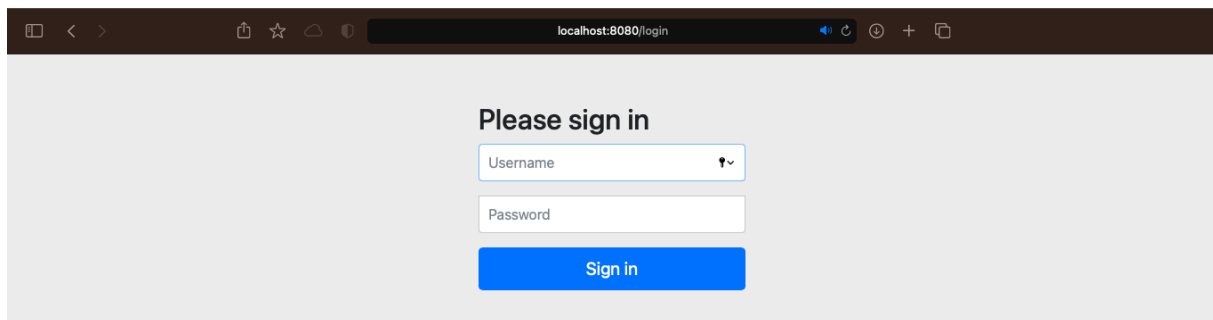
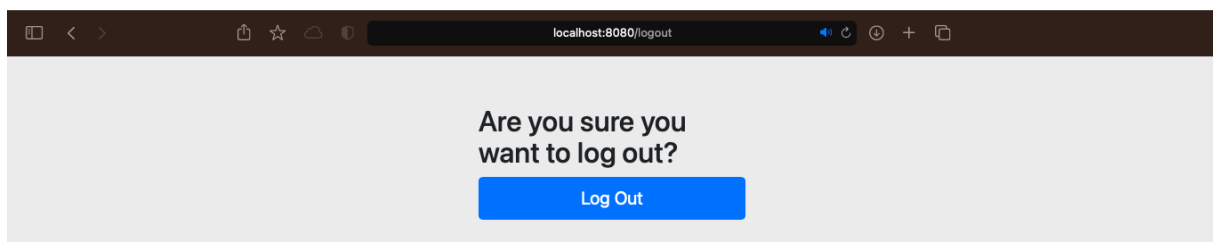


Figura 6.2: Menu do Operador.

Figura 6.3: Página de *login*.

- Logout (Figura 6.4).

Figura 6.4: Página de *logout*.

- Adicionar contas novas (Figura 6.5).

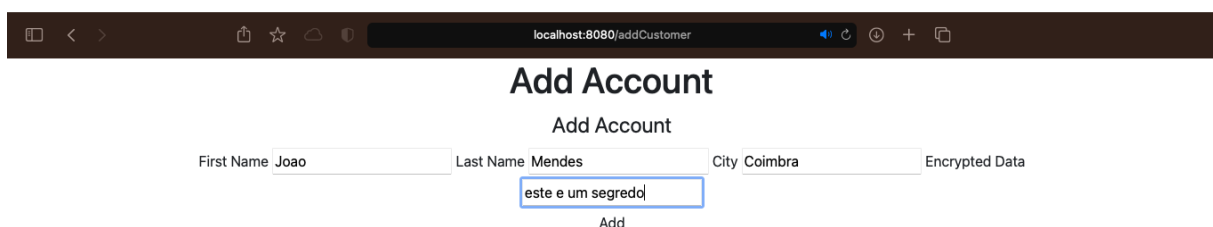
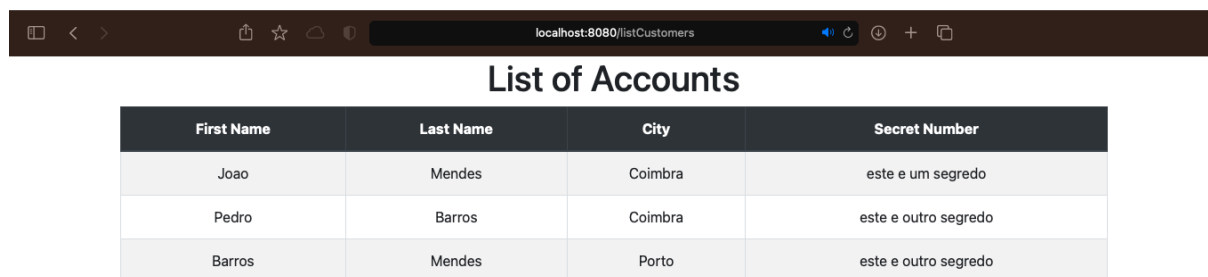


Figura 6.5: Página para adicionar contas novas.

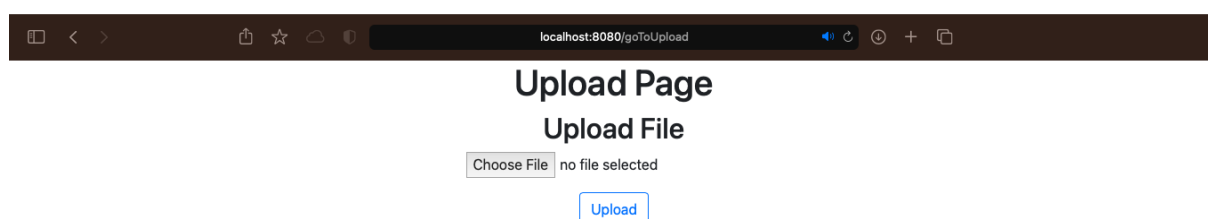
- Visualizar as conta associadas (Figura 6.6).



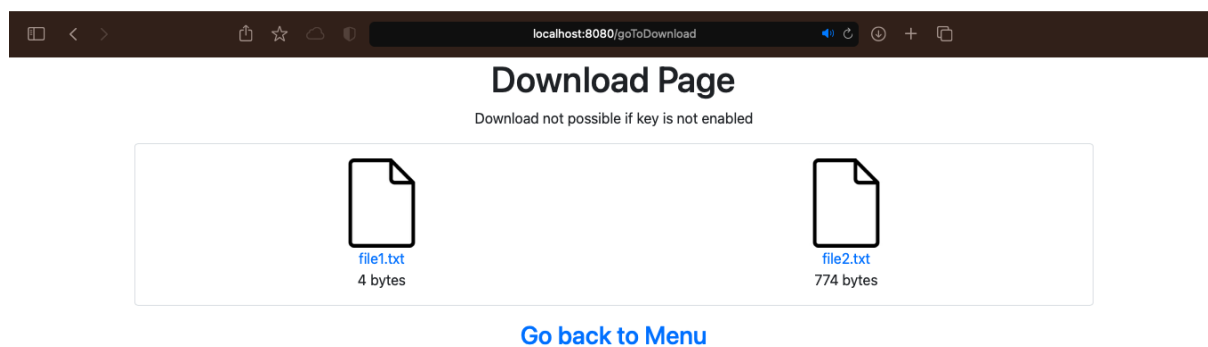
First Name	Last Name	City	Secret Number
Joao	Mendes	Coimbra	este e um segredo
Pedro	Barros	Coimbra	este e outro segredo
Barros	Mendes	Porto	este e outro segredo

Figura 6.6: Página para listar contas associadas.

- *Upload* de um ficheiro (Figura 6.7).

Figura 6.7: Página onde se pode fazer *upload* de ficheiros.

- *Download* de um ficheiro (Figura 6.8).

Figura 6.8: Página onde se pode fazer *download* de ficheiros.

6.1.1.2 JKS, OpenSSL e Processo de Cifra

Um dos pontos críticos que devem ser entendidos neste sistema é o local onde as chaves criptográficas são armazenadas. Como foi dito anteriormente, o Container Criptográfico presente na Arquitetura na secção anterior representa uma ficheiro P12, um ficheiro capaz de armazenar chaves, certificados e metadados. Este componentes são "selados" no ficheiro através de uma *password*, garantindo confidencialidade e integridade. Através da linha de comandos, pode-se

usar a ferramenta **keytool**, que permite criar chaves e/ou certificados, que serão armazenados em *keystores*.

Por razões de segurança, as chaves privadas não deverão ser exportadas da *keystore* ou de um ficheiro p12, e, portanto, o **keytool** não permite esse tipo de operações. Unicamente para se entender melhor o *software*, utilizou-se o **openssl** para exportar a chave privada do ficheiro P12. Nas cifras realizadas na Aplicação Web, nunca é usada uma chave que tenha sido exportada pelo **openssl**.

Para facilitar estes processos foi criado um *script* que permite automatizar estes processos. Apenas é criada uma *keystore* que contém as chaves e os certificados de todos os clientes, mas vários ficheiros p12, um para cada cliente. As *passwords* estão *hardcoded* no *script* e mesmo em código, algo que é uma má prática de segurança, mas sendo isto um protótipo, optou-se por essa escolha unicamente para facilitar o processo, sendo que num caso real, é uma má decisão. Pode-se então ver o *script* na Figura 6.9, e resumir-se aos seguintes passos:

1. O *script* recebe como argumento o *alias* e o Common Name. Opcionalmente, recebe uma flag **-nodes**.
2. É gerado um par de chaves RSA para o respetivo *alias* e Common Name.
3. É gerado um ficheiro P12 através da respetiva entrada da *keystore*.
4. A chave privada é exportada. Se a flag **-nodes** foi selecionada, a chave privada exportada não será protegida por 3DES.
5. O certificado é exportado.

Em resposta a pedidos feitos por um utilizador, a chave pertencente ao utilizador é acedida fazendo *load* da *keystore* na própria aplicação, e apenas depois é feito o acesso à chave. Pode-se ver um exemplo deste modo de funcionamento na Figura 6.10, em que uma chave AES é decifrada com uma chave presente na *keystore*.

Depois com essa chave AES decifrada procede-se para os processos de cifra de dados. Como tal, a coluna EncryptedData e os ficheiros são cifrados usando AES-CBC com PKCS5 Padding, de forma semelhante ao Always Encrypted. É de notar que esta dissertação foca-se em confidencialidade e decidiu-se desde o início que a integridade não deveria ser algo a ter em conta, mas que caso contrário, poderiam ser usados modos de operação que garantem integridade.

Pode-se ver como uma chave AES é usada para cifrar um ficheiro na Figura 6.11 e para decifrar um ficheiro na Figura 6.12. O IV é armazenado "à cabeça" do *array*, sabendo-se depois na decifra que serão os primeiros 16 *bytes*.

```

1  #!/bin/zsh
2  #./generate_key.sh client1 Client1
3  echo $0
4  if [ $# -lt 2 ]
5  then
6      echo "Invalid command. Do it this way :\"
7      echo \"./generate_key.sh <filename> <Owner Name>\"
8      exit 1
9  fi
10
11 DES=1
12 if [ $# -gt 2 ]
13 then
14     if [ \"$3\" -eq \"-nodes\" ]
15     then
16         DES=0
17     else
18         DES=1
19     fi
20 fi
21
22 #setopt extendedglob
23 #rm -v ^(.generate_key.sh|ks.jks)
24
25 #have to be the same key, otherwise keytool wont accept it
26 KEYSTORE_PASSWD='passwd'
27 PRIVATE_KEY_ENC_PASS='passwd'
28
29 # generate keys in keytool
30 keytool -genkeypair \
31 -alias $1 \
32 -keyalg RSA \
33 -keysize 2048 \
34 -keystore ks.jks \
35 -keypass $PRIVATE_KEY_ENC_PASS \
36 -validity 365 \
37 -storepass $KEYSTORE_PASSWD \
38 -dname \"CN=$2\" \
39 -v
40
41 #convert the jks to pkcs12
42 keytool -importkeystore -srckeystore ks.jks \
43 -srcstorepass $KEYSTORE_PASSWD \
44 -srckeypass $KEYSTORE_PASSWD \
45 -srcaalias $1 -destalias $1 \
46 -destkeystore \"p12file_$1.p12\" -deststoretype PKCS12 \
47 -deststorepass $KEYSTORE_PASSWD -destkeypass $KEYSTORE_PASSWD
48
49 if [ $DES -eq 1 ]
50 then
51     #exporting the private key with openssl from the p12 file
52     openssl pkcs12 -in \"p12file_$1.p12\" \
53     -password pass:$KEYSTORE_PASSWD \
54     -passout pass:$PRIVATE_KEY_ENC_PASS \
55     -nocerts -out private_key_$1.key
56
57     openssl pkcs12 -in p12file_$1.p12 \
58     -nocerts \
59     -password pass:passwd -passout pass:passwd \
60     | openssl rsa -passin pass:passwd
61
62 else
63     #exporting the private key with openssl from the p12 file
64     # -nodes for no DES encryption
65     openssl pkcs12 -in \"p12file_$1.p12\" \
66     -password pass:$KEYSTORE_PASSWD \
67     -info -nocerts -nodes \
68     -out private_key_$1.key
69
70     openssl pkcs12 -in p12file_$1.p12 \
71     -password pass:$KEYSTORE_PASSWD -nodes -nocerts \
72     | openssl rsa
73
74 fi
75
76 #extract only the certificate and output it
77 openssl pkcs12 -in p12file_$1.p12 -out public_cert_$1.crt -nokeys -password pass:$KEYSTORE_PASSWD
78
79 openssl x509 -in public_cert_$1.crt -noout -text
80
81
82
83
84

```

Figura 6.9: Script para gerar ficheiros P12.

```

public byte[] decipherKeyP12(byte[] cipheredKey, String username){
    try {
        //Decifra a Chave AES
        //Acede a keystore
        char[] keystorePassword = \"passwd\".toCharArray();
        char[] keyPassword = \"passwd\".toCharArray();
        KeyStore keystore = KeyStore.getInstance(\"PKCS12\");
        keystore.load(new FileInputStream( \"name: \"p12file_\" + username + \".p12\"), keystorePassword);

        //obtem a chave privada da keystore
        PrivateKey privateKey = (PrivateKey) keystore.getKey(username, keyPassword);

        //Decifra a chave AES com a chave privada RSA
        Cipher cipherRSA = Cipher.getInstance(\"RSA/ECB/OAEPWithSHA-256AndMGF1Padding\");
        cipherRSA.init(Cipher.DECRYPT_MODE, privateKey);
        return cipherRSA.doFinal(cipheredKey);
    } catch (InvalidKeyException e) {...} catch (KeyStoreException e) {...} catch (NoSuchAlgorithmException e) {...}
    return null;
}

```

Figura 6.10: Exemplo de acesso à keystore através da aplicação.

6.1.1.3 SQL Server

Foram então criados dois SQL Servers para armazenar os dados dos utilizadores e as chaves AES cifradas, um na Azure, e outro na AWS, mas não duplicados. Apesar do protótipo também pretender demonstrar a possibilidade do *provider* de serviços de base de dados ser diferente do *provider* de serviços de gestão de chaves, a conexão apenas é feita a um servidor. Portanto, nesta fase é selecionado qual o servidor pretendido, e é transparente para o resto do sistema (incluindo para a gestão de chaves criptográficas) qual o servidor que está a ser usado. A aplicação apenas deverá ter informação a que servidor se ligar, e os servidores apenas deverão ser configurados para aceitar pedidos do IP da Aplicação Web.

Pode-se ver a configuração simplificada do SQL Server na Azure na Figura 6.13 e a do SQL Server na AWS na Figura 6.14.

```

SecretKeySpec secretKeySpec = new SecretKeySpec(aesKeyBytes, "AES");

//Gera um IV aleatorio
byte[] bytesForIV=new byte[16];
SecureRandom sr = new SecureRandom();
sr.nextBytes(bytesForIV);
IvParameterSpec iv = new IvParameterSpec(bytesForIV);

//Cifra o ficheiro com a chave AES
Cipher cipherAes = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipherAes.init(Cipher.ENCRYPT_MODE, secretKeySpec,iv);
byte[] cipherTextFile=cipherAes.doFinal(fileBytes);

//concatena o IV e o resultado da cifra
byte[] finalCipher=new byte[cipherTextFile.length + bytesForIV.length];
System.arraycopy(bytesForIV, srcPos: 0, finalCipher, destPos: 0, bytesForIV.length);
System.arraycopy(cipherTextFile, srcPos: 0, finalCipher, bytesForIV.length, cipherTextFile.length);

//insere o ficheiro na base de dados
String sql = "INSERT INTO Files (ClientID,NameFile,SizeFile,CipherFile) VALUES (?, ?, ?, ?)";
PreparedStatement preparedStatement =
    connection.prepareStatement(sql);

```

Figura 6.11: Cifra de um ficheiro fornecido pelo cliente.

```

SecretKeySpec secretKeySpec = new SecretKeySpec(aesKeyBytes, "AES");

//Pesquisa o ficheiro na base de dados
String sql = "SELECT * FROM Files WHERE ClientID=? AND ID=?";
PreparedStatement preparedStatement =
    connection.prepareStatement(sql);

preparedStatement.setInt( parameterIndex: 1, userId);
preparedStatement.setInt( parameterIndex: 2, id);
ResultSet rsFile = preparedStatement.executeQuery();
if (rsFile.next()) {

    //Decifra o ficheiro cifrado com a chave AES
    Cipher cipherAes = Cipher.getInstance("AES/CBC/PKCS5Padding");
    byte[] cipherText=rsFile.getBytes( columnLabel: "CipherFile");
    byte[] bytesIv=Arrays.copyOfRange(cipherText, from: 0, to: 16);
    cipherAes.init(Cipher.DECRYPT_MODE, secretKeySpec, new IvParameterSpec(bytesIv));
    byte[] cipherTextFyleBytes=Arrays.copyOfRange(cipherText, from: 16,cipherText.length);
    byte[] plaintTextFyleBytes=cipherAes.doFinal(cipherTextFyleBytes);

    rp.setContentType("application/octet-stream");
    String headerKey="Content-Disposition";
    String headerVauLe="attachement; filename="+rsFile.getString( columnLabel: "NameFile");
    rp.setHeader(headerKey,headerVauLe);
    ServletOutputStream out = rp.getOutputStream();
    out.write(plaintTextFyleBytes);
    out.close();
}

```

Figura 6.12: Decifra de um ficheiro que o cliente pretende fazer *download*.

Para criar as tabelas na bases de dados são usados os comandos SQL presentes na Figura 6.15, seguindo a estrutura apresentada no capítulo anterior (subsecção 5.2.1.2).

Na Figura 6.16 é possível ver a resposta a *queries* feitas diretamente à base de dados (através da ferramenta Azure Data Studio). Desta forma é possível ver os dados em repouso e como a igualdade de *plaintexts* é invisível para a base de dados, pois os *plaintexts* dos EncryptedData das duas linhas do resultado da pesquisa feita sobre a tabela Accounts são iguais, assim como o conteúdo dos ficheiros da pesquisa na tabela Files.

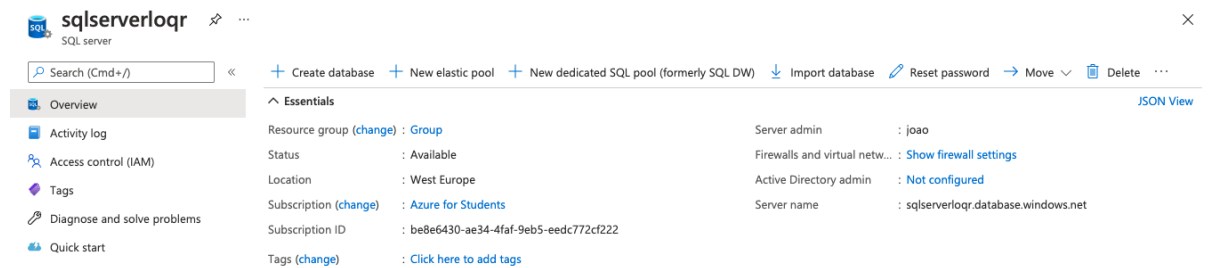


Figura 6.13: Configuração básica do SQL Server na Azure.

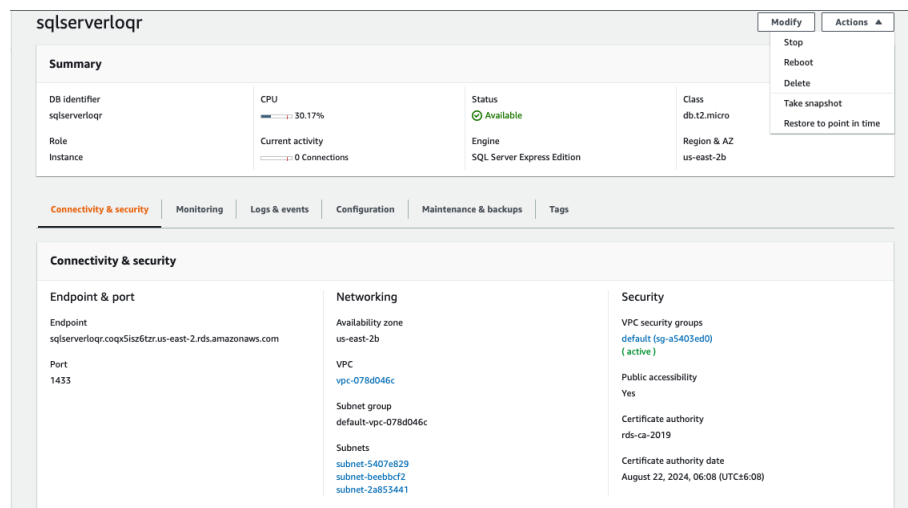


Figura 6.14: Configuração básica do SQL Server na AWS.

6.1.2 Fluxos de Dados e Ações

Nesta fase do protótipo pode-se ver os seguintes fluxos de dados e ações presentes na Figura 6.17 e a legenda nos seguintes pontos (apenas os dois primeiros pontos devem ser nessa ordem, podendo os outros ser em ordens diferentes, desde que sejam depois dos dois primeiros; os sub-pontos respeitam a ordem dentro de cada ponto):

- Inicialmente será necessário configurar o sistema para aceitar o utilizador:
 1. O Utilizador Final contacta o Operador para que consiga obter credenciais de acesso.
 2. O Operador cria as credenciais para o Utilizador e fornece-as por um outro canal (por exemplo, pessoalmente).
 3. É gerada uma entrada na *keystore* para uma chave mestra pertencente ao cliente. Também é gerada uma chave AES, que depois é cifrada com a chave presente na *keystore*. Esta chave cifrada é armazenada na base de dados.
- O Utilizador realiza o processo de autenticação necessário:
 4. O Utilizador faz o processo de *login*.

```

CREATE TABLE Clients (
    ID int IDENTITY(1,1) PRIMARY KEY,
    Username varchar(255) NOT NULL,
    Passwd varchar(255) NOT NULL,
    Role varchar(255) NOT NULL,
    Enabled TINYINT
);

CREATE TABLE Accounts (
    ID int IDENTITY(1,1) PRIMARY KEY,
    ClientID int FOREIGN KEY REFERENCES Clients(ID),
    FirstName varchar(255) NOT NULL,
    LastName varchar(255) NOT NULL,
    City varchar(255) NOT NULL,
    EncryptedData varbinary(1024) NOT NULL
);

CREATE TABLE Files (
    ID int IDENTITY(1,1) PRIMARY KEY,
    ClientID int FOREIGN KEY REFERENCES Clients(ID),
    NameFile varchar(255),
    SizeFile bigint,
    CipherFile varbinary(8000) NOT NULL
);

CREATE TABLE KeysUsed(
    ID int IDENTITY(1,1) PRIMARY KEY,
    ClientID int FOREIGN KEY REFERENCES Clients(ID),
    SecretKey varbinary(1024) NOT NULL
);

```

Figura 6.15: Comandos SQL para criar as tabelas nas bases de dados.

Run

Cancel

Disconnect

Change Connection

DBLOQR

Explain

Enable SQLCMD

Export as Notebook

1

SELECT * FROM Accounts WHERE ClientID='10'

2

SELECT * FROM Files WHERE ClientID='10'

3

SELECT * FROM Clients WHERE ID='10'

Results

Messages

	ID	ClientID	FirstName	LastName	City	EncryptedData
1	9	10	exemplo	exemplo	exemplo	0x90F23F971A3E2A5B2B389ACFDC0C6ABBD6AB25D63280956AE057A60C5E922016
2	10	10	exemplo	exemplo	exemplo	0x77BD0FA784319FC602725AECAS05FB4095C2E5021E5AF21068DE93129B510A73

	ID	ClientID	NameFile	SizeFile	CipherFile
1	6	10	exemplo.txt	8	0x91363C973315BC76E71C1B60506560759699213D9CA24CB65ACBF1E2F7229A16
2	7	10	exemplo.txt	8	0x6C788674E283069B872E30060332C0EB137A73365F55ECC3160BF49831209F23

	ID	Username	Passwd	Role	Enabled
1	10	exemplo	\$2a\$10\$0qsTOAVlhueGnAqgTd...	ROLE_USER	1

Figura 6.16: Dados em repouso na base de dados.

- Adiciona uma conta nova:
 5. O Utilizador faz um pedido para adicionar uma conta nova, contendo todos os dados necessários.
 6. A Aplicação Web seleciona qual a chave mestra pertencente ao cliente e qual a chave AES cifrada presente na base de dados.

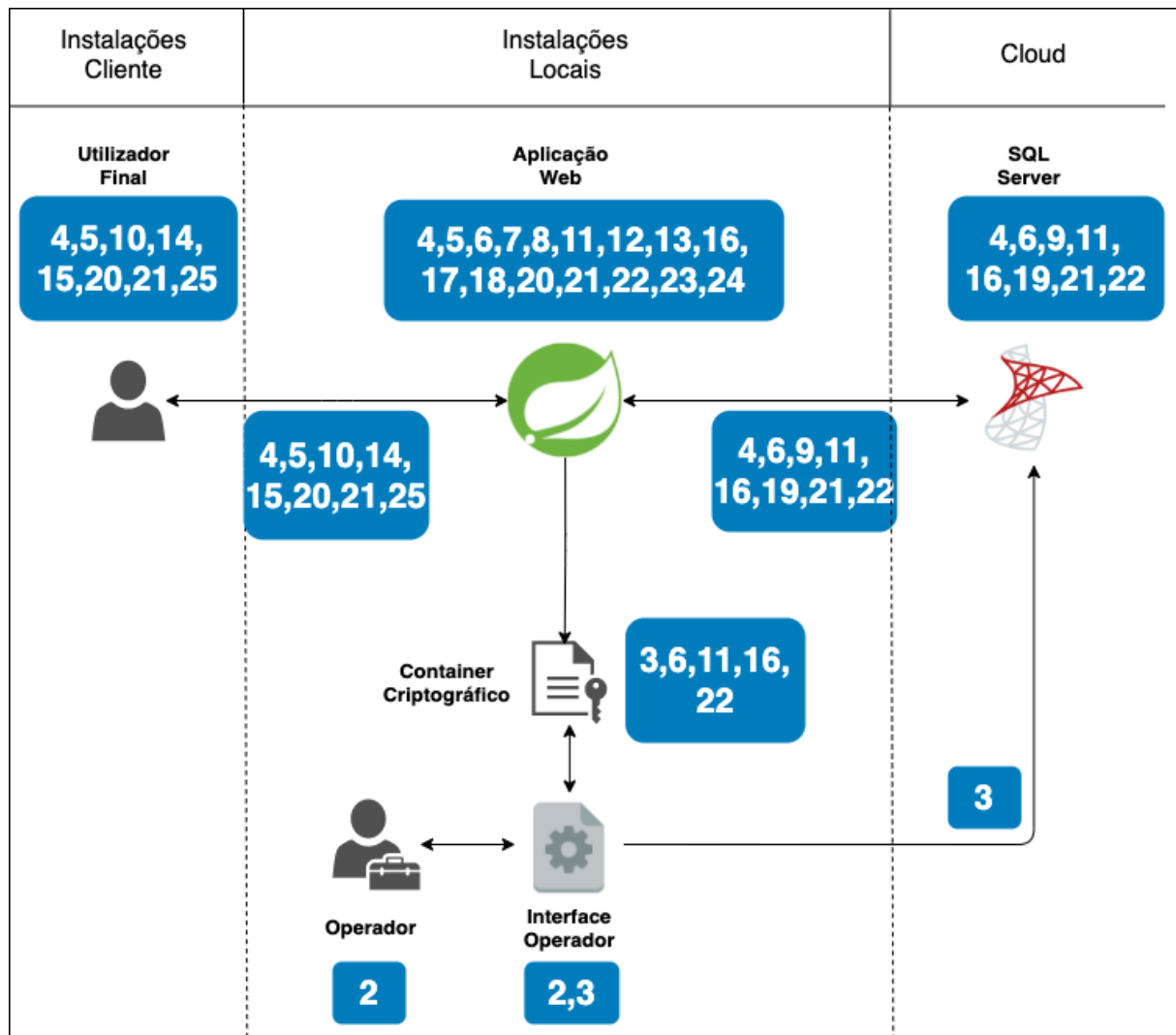


Figura 6.17: Ações e Fluxos de Dados da Primeira Fase do Protótipo.

7. Com a chave mestra decifra a chave AES cifrada e obtém a sua versão *plaintext*.
 8. Com a chave AES cifra os dados correspondentes à coluna EncryptedData.
 9. O conjunto de dados cifrados e não cifrados (colunas para além da EncryptedData) são adicionados à base de dados.
- É pretendido o acesso às informações de uma conta:
 10. O Utilizador faz um pedido para visualizar as suas contas associadas.
 11. A Aplicação Web seleciona qual a chave mestra pertencente ao cliente e qual a chave AES cifrada presente na base de dados. Também é feita uma pesquisa para obter os dados pretendidos.
 12. Com a chave mestra decifra a chave AES cifrada e obtém a sua versão *plaintext*.
 13. Com a chave AES decifra os dados correspondentes à coluna EncryptedData.
 14. Os dados são enviados em resposta ao Utilizador Final.

- O utilizador pretende armazenar um ficheiro novo.
 15. O Utilizador faz *upload* de um ficheiro.
 16. A Aplicação Web seleciona qual a chave mestra pertencente ao cliente e qual a chave AES cifrada presente na base de dados.
 17. Com a chave mestra decifra a chave AES cifrada e obtém a sua versão *plaintext*.
 18. Com a chave AES cifra o ficheiro.
 19. O ficheiro cifrado é armazenado na base de dados.
- O utilizador pretende aceder a um ficheiro.
 20. O Utilizador pretende fazer *download* de um dos seus ficheiros e envia um pedido requisitando informações de todos os seus ficheiros.
 21. A Aplicação Web seleciona os nomes e tamanhos (armazenado em *plaintext*) de ficheiros e devolve ao Utilizador.
 22. A Aplicação Web seleciona qual a chave mestra pertencente ao cliente e qual a chave AES cifrada presente na base de dados. Também é feita uma pesquisa para obter o ficheiro pretendido.
 23. Com a chave mestra decifra a chave AES cifrada e obtém a sua versão *plaintext*.
 24. Com a chave AES decifra o ficheiro.
 25. O ficheiro decifrado é enviado ao Utilizador Final.

6.2 Segunda Fase - Chaves no Key Vault e no KMS

Nesta secção serão descritas operações nos serviços de gestão de chaves criptográficas, sendo algumas necessárias para que o sistema funcione corretamente e que os requisitos sejam cumpridos, e outras adicionais apenas para fins de monitorização.

Nos processos de monitorização são considerados o número de pedidos feitos ou operações criptográficas. Sendo que para cada cliente existe apenas uma chave simétrica, um atacante que obtenha acesso aos serviços da LOQR apenas necessitaria de decifrar a chave assimétrica uma vez, e não agir de acordo com o procedimento normal. Caso esta funcionalidade fosse pretendida, poderiam ser criadas chaves simétricas para cada linha de uma tabela, de modo que para n objectos (das tabelas Accounts e Files) existiriam n chaves, fazendo com que alarmes fossem acionados na tentativa de se aceder a todos os *plaintext* dos n objetos, dado que teriam de ser feitos n pedidos de decifra de chaves aos serviços de gestão de chaves. Em contrapartida, existiria uma latência superior e o custo para o cliente seria superior, dado mais pedidos seriam feitos ao seu serviço de gestão de chaves.

6.2.1 Arquitetura, Fluxos de Dados e Ações

Dado que esta fase do protótipo representa a solução proposta na secção 5.2, a Arquitetura está presente na secção 5.2.2 e os Fluxos de Dados e Acções na secção 5.2.3.

No entanto, o seguinte componente foi adicionado durante o desenvolvimento do protótipo, não fazendo parte da concepção da solução. Para armazenar a informação necessária sobre o serviço o cliente possui (Key Vault, KMS ou P12) foi criado um ficheiro XML, em que é armazenado o username do utilizador, o tipo de serviço, e as credencias de acesso. Este ficheiro estará presente em *plaintext* na Aplicação Web. Não sendo o que se pretende demonstrar nesta dissertação, num caso real, medidas extra deverão ser tomadas em relação à sua segurança. Pode-se ver o ficheiro na Figura 6.18.

6.2.2 KMS

Aqui serão descritas certas ações e pormenores relativamente ao serviço de chaves criptográficas da Amazon Web Services.

6.2.2.1 Criação de chaves

Nesta ferramenta é possível criar chaves simétricas e assimétricas. Neste protótipo foram criadas chaves RSA (Figura 6.19). Contrariamente ao Key Vault, aqui o acesso é cedido ao nível da chave, permitindo atribuir diferentes permissões administrativas e de uso (Figura 6.20) a diferentes chaves no mesmo ambiente. Para tal, é necessário que o cliente crie um *Identity and Access*

```
<?xml version="1.0" encoding="UTF-8"?>
<clients>
  <client username="b1">
    <p12/>
  </client>
  <client username="b2">
    <p12/>
  </client>
  <client username="b3">
    <azure>
      <appId>fde75a7d-81ba-4fa5-ab71-ef5419057db1</appId>
      <password>i3urQPpPu_g95TLp7YyYkK3f_6KhKeui0e</password>
      <tenantId>42e14414-6dc7-4a3c-a8b7-20426dacdfc1</tenantId>
      <keyId>https://keyvaultclient.vault.azure.net/keys/Client-Key/47b584480a0e4744b513bb85ba8dd</keyId>
    </azure>
  </client>
  <client username="b4">
    <aws>
      <accessKeyId>AKIAVTZSCILQI2LHX7BM</accessKeyId>
      <secretAccessKey>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</secretAccessKey>
      <keyId>arn:aws:kms:us-east-2:592273621728:key/daf54464-4502-4674-9b98-0707124da3c4</keyId>
    </aws>
  </client>
</clients>
```

Figura 6.18: Ficheiro de credenciais de acesso aos serviços.

Management (IAM) *user* para lhe poder dar acesso à chave. Depois deverá fornecer as credenciais associadas a esse *user* à LOQR. Um exemplo destas credenciais podem ser vistas na Figura 6.18.

São permitidas chaves RSA de 2048, 3072, 4096 *bits*.

Create key

Configure key

Key type [Help me choose](#)

☐ Symmetric
A single encryption key that is used for both encrypt and decrypt operations

☒ Asymmetric
A public and private key pair that can be used for encrypt/decrypt or sign/verify operations

Key usage [Help me choose](#)

☒ Encrypt and decrypt
Key pairs for public key encryption
Uses the public key for encryption and the private key for decryption.

☐ Sign and verify
Key pairs for digital signing
Uses the private key for signing and the public key for verification.

Key spec [Help me choose](#)

☒ RSA_2048
☐ RSA_3072
☐ RSA_4096

Figura 6.19: Criação de uma chave no KMS.

Define key usage permissions

This account
Select the IAM users and roles that can use the CMK in cryptographic operations. [Learn more](#)

Search:

☐	Name	Path	Type
☒	admin_loqr	/	User
☐	AWSServiceRoleForRDS	/aws-service-role/rds.amazonaws.com/	Role
☐	AWSServiceRoleForSupport	/aws-service-role/support.amazonaws.com/	Role
☐	AWSServiceRoleForTrustedAdvisor	/aws-service-role/trustedadvisor.amazonaws.com/	Role
☐	kms_trail	/service-role/	Role
☐	rds-monitoring-role	/	Role

Other AWS accounts
Specify the AWS accounts that can use this key. Administrators of the accounts you specify are responsible for managing the permissions that allow their IAM users and roles to use this key. [Learn more](#)

Figura 6.20: Dar permissões de uso a um utilizador no KMS.

6.2.2.2 Código de cifra e decifra

Como foi dito anteriormente, a cifração/decifração de dados recorrendo a algoritmos baseados em AES mantém-se igual à primeira fase do protótipo. No entanto, a cifração/decifração de chaves AES foi alterada, pois agora será feita pelo KMS. Pode-se ver na Figura 6.21 e na Figura 6.22 pedidos feitos ao KMS para cifrar e decifrar chaves AES, respetivamente. Como se trata de um par de chaves assimétrico, uma alternativa seria fazer a cifra do lado da Aplicação Web com a chave pública. O mesmo já não pode acontecer com a chave privada, nem com chaves simétricas.

6.2.2.3 Impossibilitar o acesso a dados

O Operador Cliente tendo acesso à consola consegue fazer *enable/disable* das chaves mestras que lhe pertençam.

Fazendo *disable* a uma chave impossibilita-se o acesso da Aplicação Web a essa chave, não sendo assim possível decifrar a chave AES cifrada, resultando em que os dados na coluna EncryptedData e os ficheiros se tornem inacessíveis (em *plaintext*). Pode-se ver na Figura 6.23 como fazer *disable* de uma chave, na Figura 6.24 o resultado do lado da Aplicação Web, e na Figura 6.25 o que é apresentado ao Utilizador Final no seu *browser* quando este tenta aceder aos seus dados guardados em tabelas ou fazer *download* de um ficheiro, sendo que a chave mestra foi *disabled*.

Para reverter o processo anterior, a chave apenas deverá ser *enabled*, não tendo sido

```

public static byte[] generateAESKeyAndCipherItAWS(String username,
                                                String accessKeyId,
                                                String secretAccessKey,
                                                String keyId){
    try {
        AWSKMS kmsClient = AWSKMSClientBuilder.standard().withRegion(Regions.US_EAST_2).
            withCredentials(new AWSStaticCredentialsProvider(
                new BasicAWSCredentials(
                    accessKeyId,
                    secretAccessKey))).
            build();

        //Adiciona o Provider
        Security.addProvider(new BouncyCastleProvider());

        // Gera uma chave AES
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(128);
        SecretKey secretKey = keyGen.generateKey();

        EncryptRequest req = new EncryptRequest().withKeyId(keyId).
            withEncryptionAlgorithm("RSAES_OAEP_SHA_256").
            withPlaintext(ByteBuffer.wrap(secretKey.getEncoded()));
        com.amazonaws.services.kms.model.EncryptResult ciphertext = kmsClient.encrypt(req);
        return ciphertext.getCiphertextBlob().array();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    return null;
}

```

Figura 6.21: Cifra de uma chave AES na segunda fase do protótipo, utilizando pedidos feitos ao KMS.

```

public byte[] decipherKeyAws(byte[] cipheredKey, ArrayList<String> credentials){
    try{
        String accessKeyId = credentials.get(0);
        String secretAccessKey = credentials.get(1);
        String keyId = credentials.get(2);
        //decifra a chave AES que foi cifrada com uma chave na KMS
        AWSKMS kmsClient = AWSKMSClientBuilder.standard().withRegion(Regions.US_EAST_2).
            withCredentials(new AWSStaticCredentialsProvider(
                new BasicAWSCredentials(accessKeyId, secretAccessKey))).
            build();

        ByteBuffer cipherText = ByteBuffer.wrap(cipheredKey);
        DecryptRequest req = new DecryptRequest().withCiphertextBlob(cipherText).
            withKeyId(keyId).withEncryptionAlgorithm("RSAES_OAEP_SHA_256");
        return kmsClient.decrypt(req).getPlaintext().array();
    } catch (com.amazonaws.services.kms.model.DisabledException except) {
        //a chave foi disabled pelo cliente
        return null;
    }
}

```

Figura 6.22: Decifra de uma chave AES na segunda fase do protótipo, utilizando pedidos feitos ao KMS.

comprometidos quaisquer dados.

6.2.2.4 Monitorização

Apesar da monitorização de acesso às chaves não ser um dos requisitos pretendidos, adiciona funcionalidades interessantes, dado que pode permitir ver se existe uma escassez de acessos a chaves, ou uma demasia.

Através da ferramenta CloudWatch (também pertencente à AWS), é possível criar um gráfico onde se podem visualizar várias métricas, e definir alarmes para quando um determinado limite de

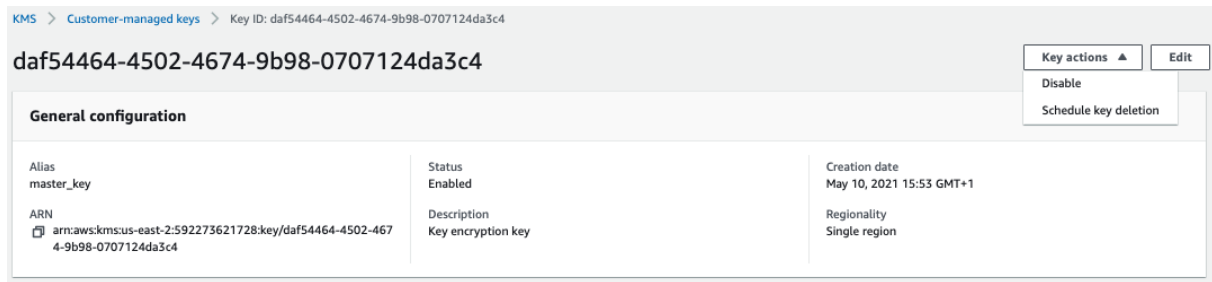


Figura 6.23: Como fazer *disable* a uma chave através da console do KMS.

```
2021-07-20 19:50:36.015 ERROR 78576 --- [nio-8080-exec-3] o.a.c.c.c.[.][.].dispatcherServlet : Servlet.service() for servlet [dispatcherServlet]
com.amazonaws.services.kms.model.DisabledException: arn:aws:kms:us-east-2:592273621728:key/daf54464-4502-4674-9b98-0707124da3c4 is disabled. (Service: AmazonKeyManagementService, Status: 400, RequestID: 11111111-1111-1111-1111-111111111111)
at com.amazonaws.http.AmazonHttpClient$RequestExecutor.handleErrorResponse(AmazonHttpClient.java:1811) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient$RequestExecutor.handleServiceErrorResponse(AmazonHttpClient.java:1395) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient$RequestExecutor.executeOneRequest(AmazonHttpClient.java:1371) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient$RequestExecutor.executeHelper(AmazonHttpClient.java:1145) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient$RequestExecutor.doExecute(AmazonHttpClient.java:802) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient$RequestExecutor.executeWithTimer(AmazonHttpClient.java:770) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient$RequestExecutor.execute(AmazonHttpClient.java:744) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient$RequestExecutor.access$500(AmazonHttpClient.java:784) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient$RequestExecutionBuilderImpl.execute(AmazonHttpClient.java:686) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient.execute(AmazonHttpClient.java:550) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.http.AmazonHttpClient.execute(AmazonHttpClient.java:530) ~[aws-java-sdk-core-1.11.792.jar:na]
at com.amazonaws.services.kms.AWSKMSClient.doInvoke(AWSKMSClient.java:5831) ~[aws-java-sdk-kms-1.11.792.jar:na]
at com.amazonaws.services.kms.AWSKMSClient.invoke(AWSKMSClient.java:5798) ~[aws-java-sdk-kms-1.11.792.jar:na]
at com.amazonaws.services.kms.AWSKMSClient.invoke(AWSKMSClient.java:5787) ~[aws-java-sdk-kms-1.11.792.jar:na]
at com.amazonaws.services.kms.AWSKMSClient.executeDecrypt(AWSKMSClient.java:1582) ~[aws-java-sdk-kms-1.11.792.jar:na]
at com.amazonaws.services.kms.AWSKMSClient.decrypt(AWSKMSClient.java:1553) ~[aws-java-sdk-kms-1.11.792.jar:na]
at net.app.webapp.AppController.decipherKeyAws(AppController.java:637) ~[classes/:na]
at net.app.webapp.AppController.getKey(AppController.java:498) ~[classes/:na]
at net.app.webapp.AppController.listCustomers(AppController.java:294) ~[classes/:na] <14 internal calls>
```

Figura 6.24: *Print* da exceção na Aplicação Web.

acessos a uma chave é ultrapassado. É possível configurar várias opções para receber notificações de tais acontecimentos. Desta forma, será possível que o Operador Cliente consiga ter uma resposta mais rápida a uma série de acontecimentos inesperados, e tomar ações em função de tal, sem qualquer interação da LOQR. Um exemplo seria fazer *disable* a uma chave que tenha sido usada um número de vezes bastante superior ao achado normal.

Usando esta ferramenta é possível:

- Criar um *dashboard* para ver os acesso feitos às chaves (Figura 6.26).
- Configurar um alarme (Figura 6.27):

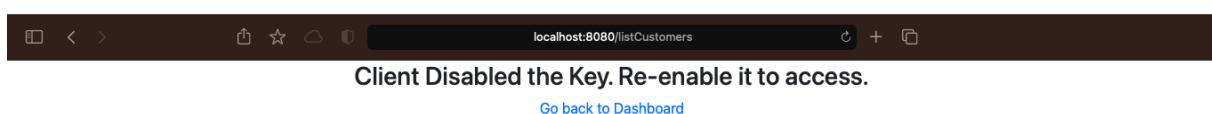


Figura 6.25: Resultado que o Cliente visualiza depois de a chave ser *disable*.

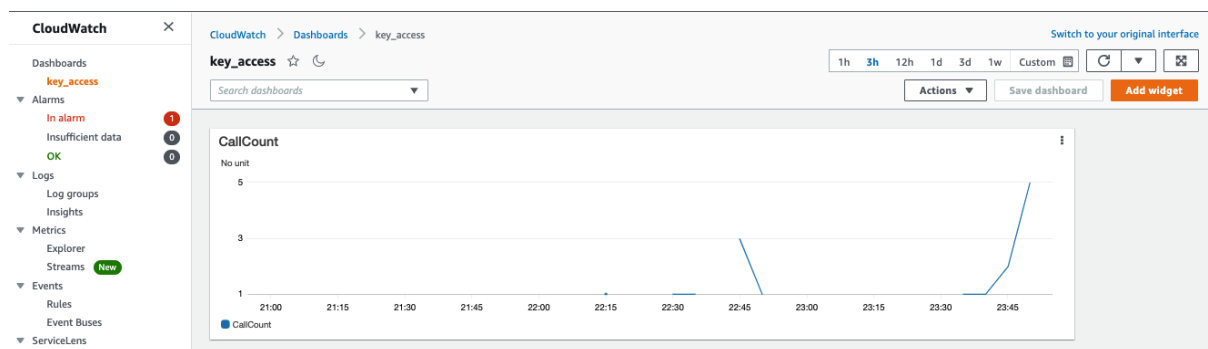


Figura 6.26: Criação de um *dashboard* para ver acessos a chaves no KMS.

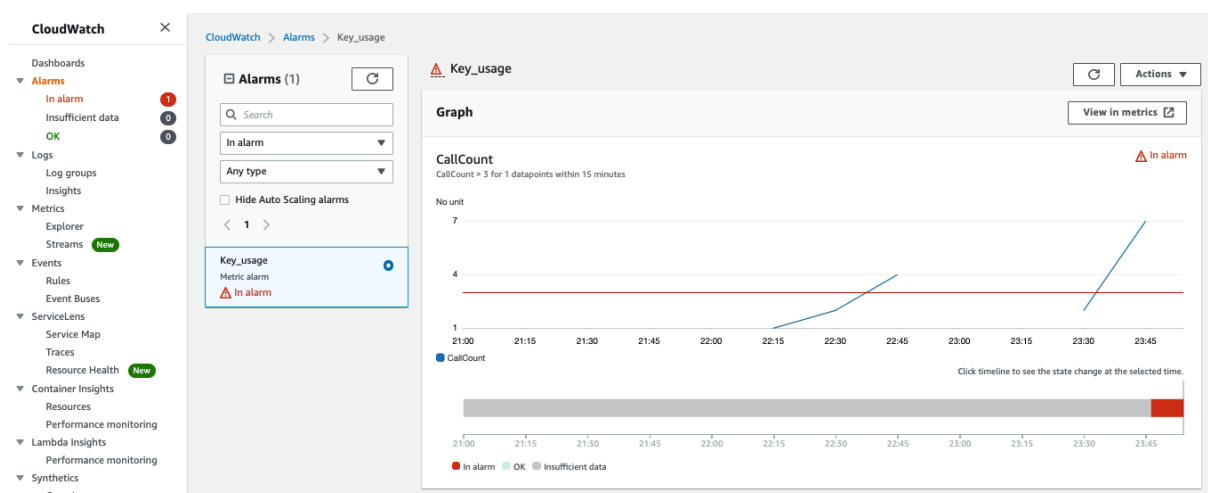


Figura 6.27: Configurar um alarme no KMS.

1. Definir a métrica que deve ser tomada em conta para o alarme (Figura 6.28). Por exemplo: "Número de operações criptográficas no espaço de 15 minutos". No entanto, não foi conseguido diferenciar o tipo de operação criptográfica.

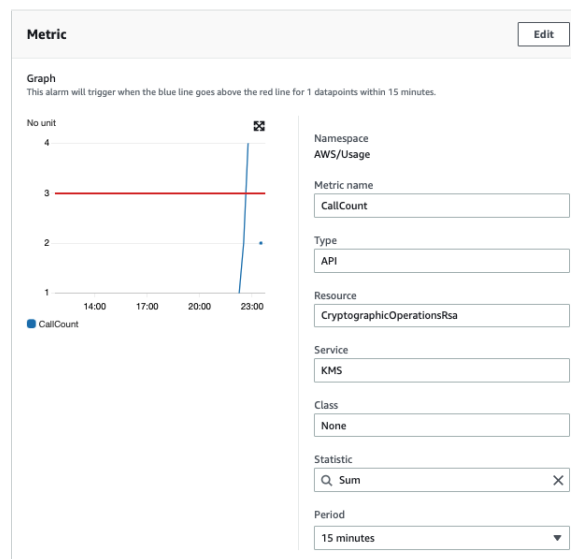


Figura 6.28: Definir a métrica que deve ser observada num alarme no KMS.

2. Escolher qual a condição para que o alarme seja ativado (Figura 6.29).

The screenshot shows the 'Conditions' configuration interface in AWS CloudWatch. The 'Threshold type' section has 'Static' selected. The 'Whenever CallCount is...' section has 'Greater' selected. The 'than...' section shows the threshold value '3'. The 'Additional configuration' section is collapsed.

Threshold type

- ☒ Static
- ☐ Anomaly detection

Whenever CallCount is...

- ☒ Greater
- ☐ Greater/Equal
- ☐ Lower/Equal
- ☐ Lower

than...

Define the threshold value.

3

Must be a number

► Additional configuration

Figura 6.29: Definir a condição para um alarme ser ativado no KMS.

3. Selecionar qual o tipo de notificação (Figura 6.30).
4. Receber um e-mail de quando a condição é ativada (Figura 6.31).
5. Receber e-mails com *logs* das atividades posteriores à condição ser ativada (Figura 6.32).

Figura 6.30: Selecionar o tipo de notificação associado a um alarme no KMS.

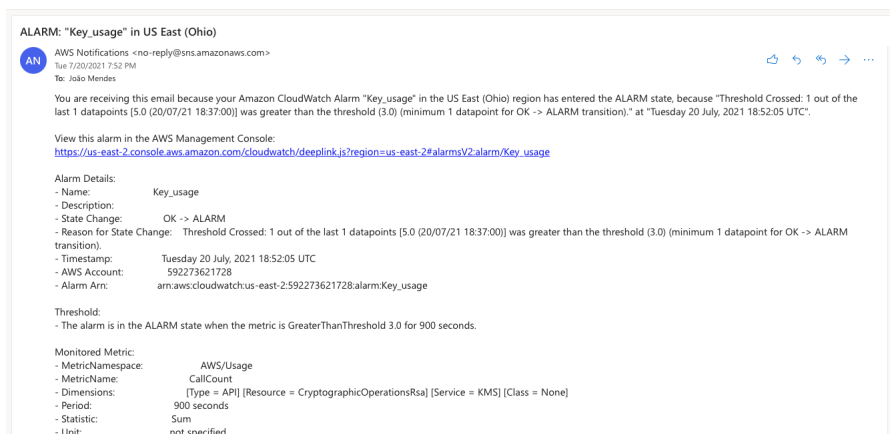


Figura 6.31: Exemplo de um e-mail recebido após um alarme ser ativado no KMS.

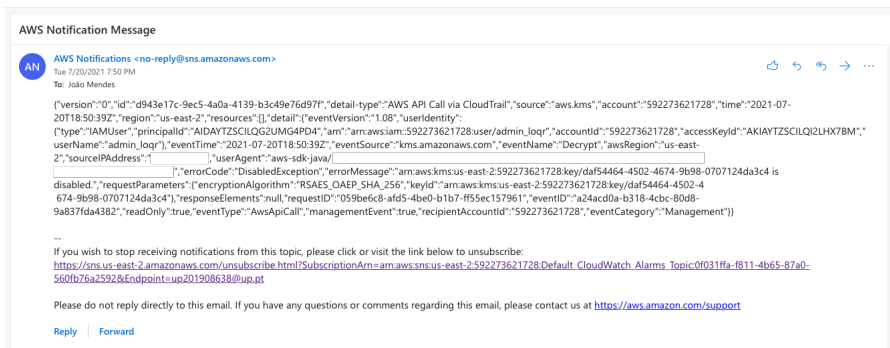


Figura 6.32: Exemplo de um e-mail com logs de atividades ocorridas depois de um alarme ter sido ativado no KMS.

6.2.3 Key Vault

Aqui serão descritas certas ações e pormenores relativamente ao serviço de chaves criptográficas da Azure.

6.2.3.1 Criação de chaves

O Key Vault funciona de forma diferente ao KMS, no sentido em que o ambiente em que as chaves residem pode não ser partilhado, ou seja, podem ser criados vários Key Vaults, cada um com as suas chaves.

Nesta ferramenta é possível criar chaves assimétricas, não sendo possível criar chaves simétricas. Neste protótipo apenas foram criadas chaves RSA (Figura 6.33). Contrariamente ao KMS, aqui o acesso é cedido ao nível do KeyVault, permitindo atribuir diferentes permissões administrativas e de uso (Figura 6.34) ao conjunto de todas as chaves num determinado Key Vault. Para tal, é necessário que o cliente registre uma Aplicação e que lhe dê acesso ao Key Vault, através da ferramenta Cloud Shell pertencente ao portal da Azure (Figura 6.35). Depois deverá fornecer as credencias associadas a essa Aplicação à LOQR, obtidas através dos comandos executados no Cloud Shell. Um exemplo destas credenciais podem ser vistas na Figura 6.18.

São permitidas chaves RSA de 2048, 3072, 4096 *bits*.

Home > KeyVaultClient >

Create a key ...

Options	Generate
Name *	example-key
Key type	☒ RSA ☐ EC
RSA key size	☒ 2048 ☐ 3072 ☐ 4096
Set activation date	☐
Set expiration date	☐
Enabled	<input checked="" type="button"/> Yes No

Figura 6.33: Criação de uma chave num Key Vault.

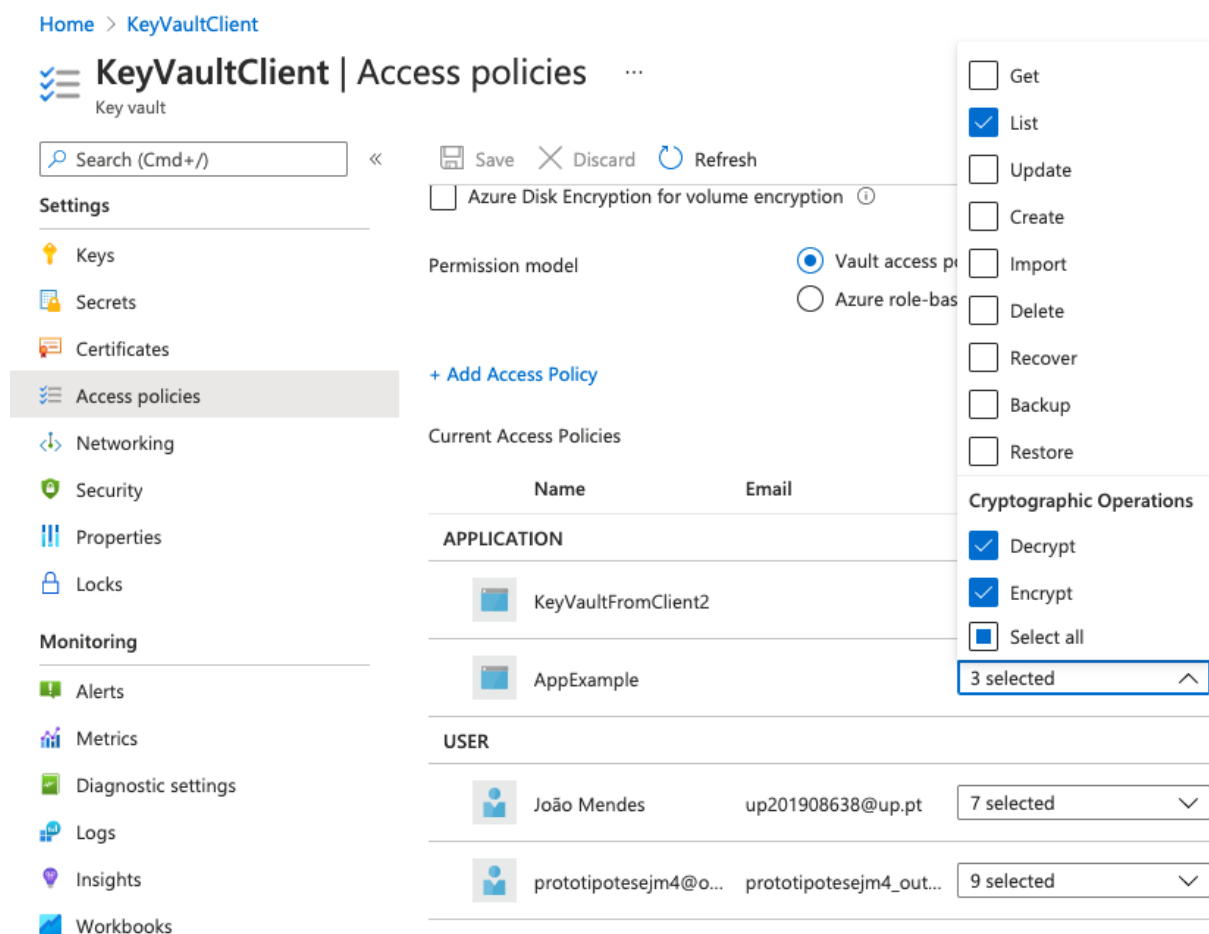


Figura 6.34: Definir permissões de acesso a uma Key Vault.

```
prototipotesejm4@Azure:~$ az ad sp create-for-ibac -n AppExample --skip-assignment
The output includes credentials that you must protect. Be sure that you do not include these credentials in your code or check the credentials into your source code.
See https://aka.ms/azadsp-cli
'name' property in the output is deprecated and will be removed in the future. Use 'appId' instead.
{
  "appId": "3bfdeffa-6ccc-4117-a8a6-c147ded8a243",
  "displayName": "AppExample",
  "name": "3bfdeffa-6ccc-4117-a8a6-c147ded8a243",
  "password": "L6h-eOi.-lWZ45HTQkptoSnNwksZPy~_z",
  "tenant": "42e14141-6dc7-4a3c-a8b7-20426dadfc1"
}
prototipotesejm4@Azure:~$ az keyvault set-policy --name KeyVaultClient --spn "3bfdeffa-6ccc-4117-a8a6-c147ded8a243" --key-permissions list encrypt decrypt
{
  "id": "/subscriptions/50c8f214-f0d4-41c6-a7d8-ba9a6aee9043/resourceGroups/Group/providers/Microsoft.KeyVault/vaults/KeyVaultClient",
  "location": "westeurope",
  "name": "KeyVaultClient",
  "permissions": [
    {
      "permissions": [
        "list",
        "encrypt",
        "decrypt"
      ],
      "tenantId": "42e14141-6dc7-4a3c-a8b7-20426dadfc1"
    }
  ]
}
```

Figura 6.35: Registo e atribuição de permissões através do Cloud Shell.

6.2.3.2 Código de cifra e decifra

Mais uma vez, nesta fase do protótipo, a cifração/decifração de dados recorrendo a algoritmos baseados em AES mantém-se igual à primeira fase do protótipo. No entanto, a cifração/decifração de chaves AES foi alterada, pois agora será feita pelo Key Vault. Pode-se ver na Figura 6.36 e na Figura 6.37 pedidos feitos ao Key Vault para cifrar e decifrar chaves AES, respetivamente. Como se trata de um par de chaves assimétrico, uma alternativa seria fazer a cifra do lado da Aplicação web com a chave pública. O mesmo já não pode acontecer com a chave privada.

```
public static byte[] generateAESKeyAndCipherItAzure(String username, String appId, String password, String tenantId, String keyId){
    try {
        // Usa as credencias fornecidas pelo user
        ClientSecretCredential credentials = new ClientSecretCredentialBuilder()
            .tenantId(tenantId)
            .clientId(appId)
            .clientSecret(password)
            .build();

        //Cria um crypto client da biblioteca da Azure
        CryptographyClient cryptoClient = new CryptographyClientBuilder()
            .credential(credentials)
            .keyIdentifier(keyId)
            .buildClient();

        //Adiciona o Provider
        Security.addProvider(new BouncyCastleProvider());

        // Gera uma chave AES
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init( keySize: 128);
        SecretKey secretKey = keyGen.generateKey();

        // Funciona se a key estiver Disabled
        EncryptResult encryptionResult = cryptoClient.encrypt(EncryptionAlgorithm.RSA_OAEP_256, secretKey.getEncoded() );
        return encryptionResult.getCipherText();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    return null;
}
```

Figura 6.36: Cifra de uma chave AES na segunda fase do protótipo, utilizando pedidos feitos ao Key Vault.

```
public byte[] decipherKeyAzure(byte[] cipheredKey, ArrayList<String> listCredentials){
    try {
        String appId = listCredentials.get(0);
        String passwd = listCredentials.get(1);
        String tenantId = listCredentials.get(2);
        String keyId = listCredentials.get(3);

        ClientSecretCredential credentials = new ClientSecretCredentialBuilder()
            .tenantId(tenantId)
            .clientId(appId)
            .clientSecret(passwd)
            .build();

        CryptographyClient cryptoClient = new CryptographyClientBuilder()
            .credential(credentials)
            .keyIdentifier(keyId)
            .buildClient();

        DecryptResult decryptionResult = cryptoClient.
            decrypt(EncryptionAlgorithm.RSA_OAEP_256, cipheredKey);
        return decryptionResult.getPlainText();
    } catch (com.azure.core.exception.HttpResponseException e) {
        if(e.getResponse().getStatusCode()==403){
            //Forbidden access -> chave disabled
            return null;
        }
    }
    return null;
}
```

Figura 6.37: Decifra de uma chave AES na segunda fase do protótipo, utilizando pedidos feitos ao Key Vault.

6.2.3.3 Impossibilitar o acesso a dados

O Operador Cliente tendo acesso ao portal da Azure consegue fazer *enable/disable* das chaves mestras que lhe pertençam.

Fazendo *disable* a uma chave impossibilita-se o acesso da Aplicação Web a essa chave, não sendo assim possível decifrar a chave AES cifrada, resultando em que os dados na coluna EncryptedData e os ficheiros se tornem inacessíveis (em *plaintext*). Pode-se ver na Figura 6.38 como fazer *disable* de uma chave e na Figura 6.39 o resultado do lado da Aplicação Web.

O resultado que é apresentado no *browser* quando o Utilizador Final tenta aceder a recursos que foram cifrados com uma chave AES, que foi cifrada com uma chave que posteriormente foi *disabled*, é semelhante ao do KMS (Figura 6.25).

Para reverter o processo anterior, a chave apenas deverá ser *enabled*, não tendo sido comprometidos quaisquer dados.

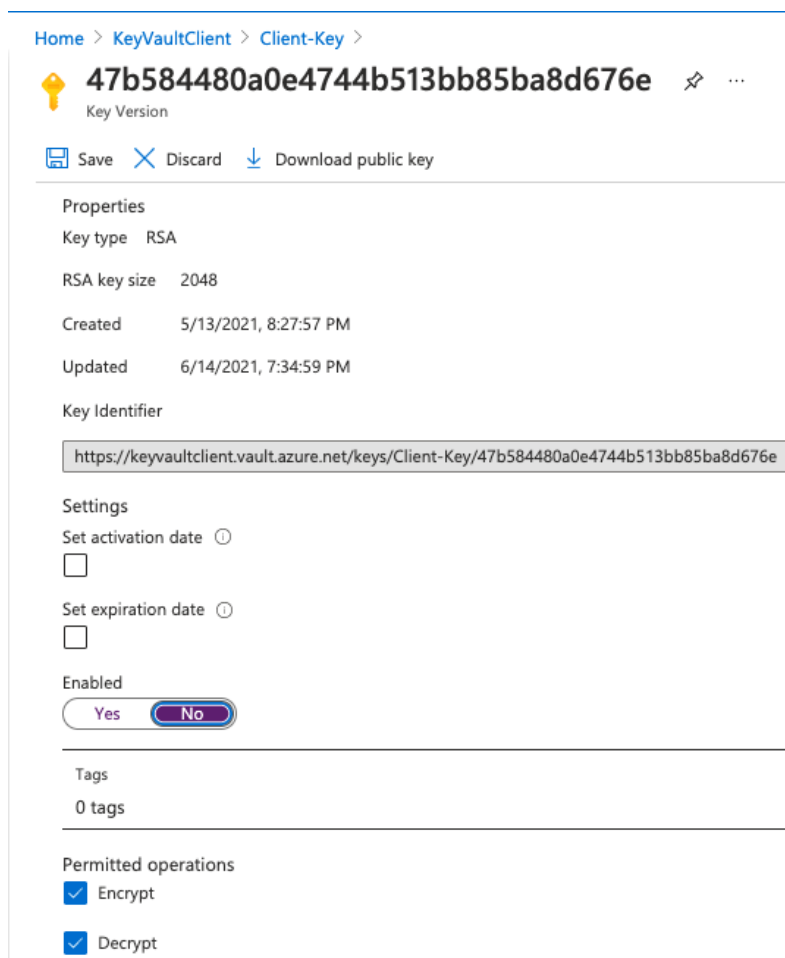


Figura 6.38: Como fazer *disable* a uma chave através do portal da Azure.

```
com.azure.core.exception.HttpResponseException: Status code 403, "{ \"error\": { \"code\": \"Forbidden\", \"message\": \"Operation decrypt is not allowed on a disabled key.\" }, \"innererror\": { \"code\": \"KeyDisabled\" } }\" <3 internal calls>
```

Figura 6.39: *Print* da exceção na Aplicação Web.

6.2.3.4 Monitorização

De forma semelhante aos serviços da AWS, é possível adicionar alarmes de quando certos limites de número de acessos são ultrapassados. Como foi dito anteriormente, apesar de não ser possível fazer um *disable* automático de uma determinada chave, é possível reagir mais rapidamente a comportamentos fora do normal.

Acedendo às opções "Metrics" e "Alerts" no menu de um Key Vault é possível:

- Monitorizar métricas relativamente aos acessos feitos num determinado espaço de tempo e visualizar num gráfico (Figura 6.40).

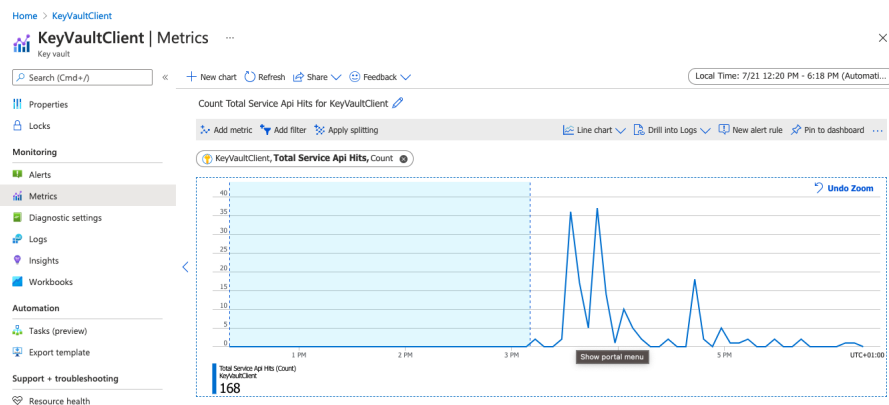


Figura 6.40: Monitorizar métricas de um Key Vault.

- Definir alarmes:
 1. Criar um grupo que pode ser associado ao alarme (Figura 6.41), e adicionar e-mails a esse grupo (Figura 6.42). Existem vários tipos de notificações possíveis, mas optou-se por e-mail.
 2. Definir a condição para o alarme ser ativado (Figura 6.43).
 3. Atribuir definições adicionais, como o nome do alarme, descrição e um nível de severidade de 0 a 4 (Figura 6.44).
 4. Receber um e-mail quando um alarme é ativado (Figura 6.45).

Create action group ...

Basics Notifications Actions Tags Review + create

An action group invokes a defined set of notifications and actions when an alert is triggered. [Learn more](#)

Project details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *

Resource group * [Create new](#)

Instance details

Action group name *

Figura 6.41: Criar um grupo para ser associado a um alarme de um Key Vault.

Home > KeyVaultClient > Manage actions >

Create action group ...

Basics Notifications Actions Tags Review + create

Notifications

Configure the method in which users will be notified when the action group triggers. Select notification types, provide receiver details and add a unique description. This step is optional.

Notification type	Name	Selected
Email/SMS message/Push/Voice	up201908638@up.pt	☒ Email
		☐

Email/SMS message/Push/Voice

Add or edit an Email/SMS/Push/Voice action

☒ Email

Email *

☐ SMS (Carrier charges may apply)

Country code

Phone number

☐ Azure app Push Notifications

Azure account email

Figura 6.42: Adicionar um mail a um grupo de um alarme de um Key Vault.

Alert logic

Threshold ☐ Static ☒ Dynamic

Operator Aggregation type * Threshold value * Unit *

Condition preview

Whenever the count of total service api hits is greater than 3

Evaluated based on

Aggregation granularity (Period) * Frequency of evaluation

Figura 6.43: Definir a condição para um alarme de um Key Vault ser ativado.

Alert rule details

Provide details on your alert rule so that you can identify and manage it later.

Alert rule name *

Description

Subscription

Resource group *

Severity *

Enable alert rule upon creation ☒

Automatically resolve alerts ☒

Figura 6.44: Atribuir definições adicionais a uma alarme de um Key Vault.

☐ Your Azure Monitor alert was triggered

Azure monitor alert rule alert_key_condition was triggered for keyvaultclient at July 21, 2021 14:31 UTC.

Alert rule description	Number of key accesses exceeded
Rule ID	/subscriptions/50c8f214-f0d4-41c6-a7d8-ba9a6aee9043/resourceGroups/Group/providers/microsoft.insights/metricAlerts/alert_key_condition View Rule >
Resource ID	/subscriptions/50c8f214-f0d4-41c6-a7d8-ba9a6aee9043/resourceGroups/Group/providers/microsoft.keyvault/vaults/keyvaultclient View Resource >

Alert Activated Because:

Metric name	ServiceApiHit
Metric namespace	vaults/keyvaultclient
Dimensions	microsoft.resourceId = /subscriptions/50c8f214-f0d4-41c6-a7d8-ba9a6aee9043/resourceGroups/Group/providers/microsoft.keyvault/vaults/keyvaultclient
Time Aggregation	Count
Period	Over the last 5 mins
Value	11
Operator	GreaterThan
Threshold	3
Criterion Type	StaticThresholdCriterion

Figura 6.45: Exemplo de um e-mail de quando um alarme é ativado.

Capítulo 7

Conclusão

Esta dissertação começou com tendo em conta um serviço oferecido a clientes da LOQR que já permitia segurança contra um atacante com acesso à base de dados. No modelo de confiança do respetivo serviço assumia-se que o responsável de segurança por parte da LOQR era honesto, e que a chave mestra não era cedida a agentes externos. Também era esse mesmo responsável que teria de ter em consideração quais os processos a ter em relação à chave, caso informações da base de dados fossem adquiridas por terceiros ou divulgadas publicamente. Portanto, nesse serviço, o cliente teria de assumir que a LOQR cumpriria com as suas responsabilidades em relação às chaves criptográficas.

Como tal, começou-se por se investigar soluções propostas que visam solucionar problemas semelhantes. Apesar de nenhuma dessas soluções se apresentar como uma solução direta para o que se pretendia nesta dissertação, apresentam sistemas que têm componentes e propriedades que poderiam ser adaptadas para alcançar os objetivos pretendidos.

Posteriormente, analisou-se os componentes do sistema do serviço da LOQR, para ser possível identificar os requisitos pretendidos, e em seguida propôs-se uma solução que tinha como objetivo alcançar esses requisitos. Foi fundamentado o porquê de a solução proposta cumprir os tais requisitos, sendo feita uma ligação direta entre os requisitos e a respetiva justificação. Em seguida, foi desenvolvido um protótipo que demonstra pormenores técnicos das principais tecnologias, os processos de cifração/decifração, comportamentos nas plataformas de serviços de gestão de chaves, e como estes alteram acessos a dados do sistema.

Adicionalmente foram adicionados serviços de notificações que permitem identificar comportamentos anormais no número de acessos a chaves criptográficas, o que pode ajudar a reconhecer possíveis falhas de segurança.

Conclui-se assim que os objetivos desta dissertação foram alcançados, demonstrando que o sistema do serviço da LOQR pode ser adaptado para dar controlo da chave aos seus clientes, mantendo a possibilidade de clientes que não pretendam tais funcionalidades manter as atuais, e minimizando a dificuldade da transição, tentando preservar todos os componentes não relacionados com gestão de chaves criptográficas.

7.1 Trabalho Futuro

Soluções alternativas e mais complexas poderão ser aplicadas baseando-se na solução proposta nesta dissertação.

Nesta solução, é definida inicialmente qual é a coluna que armazena dados cifrados. Um alternativa seria cada cliente ter a possibilidade de numa tabela definir dinamicamente quais as colunas cifradas, e quais poderiam deixar de ser cifradas. Esta solução adiciona funcionalidades que permitem que o sistema seja mais dinâmico, mas não adiciona novas funcionalidades na parte de gestão de chaves criptográficas.

Também, poderiam ser criadas várias chaves simétricas para cada cliente, alternativamente a uma, de modo a que no caso em que um atacante consiga ter acesso à base de dados e às credencias de acesso a chaves criptográficas, teria de fazer um número mais elevado de pedidos aos serviços de gestão de chaves. Isto tornaria mais útil o serviço de notificações, o que permitiria uma resposta mais realista a tal incidente. No entanto, um número mais elevado de pedidos consequentes desta solução iria aumentar o custo do lado do cliente, dado que parte do custo do serviço de gestão de chaves provém do número de acessos às chaves.

Alternativamente, o Key Vault e o KMS também poderão ser hospedados em HSM's na Cloud, para garantir que as chaves não estão apenas protegidas por *software*, mas também estarão protegidas por *hardware*. Estes dois componentes estão em conformidade FIPS 140-2 de nível 2 e de nível 3. Tal solução não foi possível de desenvolver por questões de tempo.

Bibliografia

- [1] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall Press, USA, 6th edition, 2013. ISBN: 0133354695.
- [2] Quynh Dang. [Secure hash standard](#), 2015-08-04 2015. doi:<https://doi.org/10.6028/NIST.FIPS.180-4>.
- [3] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. [Advanced encryption standard \(aes\)](#), 2001-11-26 2001. doi:<https://doi.org/10.6028/NIST.FIPS.197>.
- [4] National Institute of Standards and Technology. [Recommendation for block cipher modes of operation methods and techniques](#). Technical Report Special Publication (NIST SP) 800-38A, U.S. Department of Commerce, Washington, D.C., December 2001.
- [5] National Institute of Standards and Technology. [Recommendation for block cipher modes of operation: Galois/counter mode \(gcm\) and gmac](#). Technical Report Special Publication (NIST SP) 800-38D, U.S. Department of Commerce, Washington, D.C., November 2007.
- [6] David A. McGrew and John Viega. [The galois/counter mode of operation \(gcm\)](#). 2005.
- [7] R. L. Rivest, A. Shamir, and L. Adleman. [A method for obtaining digital signatures and public-key cryptosystems](#). *Commun. ACM*, 21(2):120–126, February 1978. ISSN: 0001-0782. doi:10.1145/359340.359342.
- [8] IETF. [PKCS #1: RSA Cryptography Specifications Version 2.2](#). RFC 8017, November 2016.
- [9] IETF. [PKCS #1: RSA Cryptography Specifications Version 2.2, Section 7.1](#). RFC 8017, November 2016.
- [10] Microsoft. [Indexes](#). Online, Acedido em 2021.
- [11] Oracle. [Package java.sql](#). Online, Acedido em 2021.
- [12] Oracle. [Azure data studio](#). Online, Acedido em 2021.
- [13] AWS. [Kms](#). Online, Acedido em 2020.

- [14] Microsoft. [Key vault](#). Online, Acedido em 2020.
- [15] Microsoft. [Key vault compliance](#). Online, Acedido em 2021.
- [16] AWS. [Kms compliance](#). Online, Acedido em 2021.
- [17] Oracle. [Java cryptography architecture standard algorithm name documentation](#). Online, Acedido em 2021.
- [18] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Mitchell, and Robert Cunningham. [Sok: Cryptographically protected database search](#). pages 172–191, 05 2017. doi:10.1109/SP.2017.10.
- [19] Sumeet Bajaj and Radu Sion. [Trusteddb: A trusted hardware-based database with privacy and data confidentiality](#). volume 26, pages 205–216, 01 2011. doi:10.1145/1989323.1989346.
- [20] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. [Orthogonal security with cipherbase](#). In *6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*, January 2013.
- [21] Christian Priebe, Kapil Vaswani, and Manuel Costa. [Enclavedb: A secure database using sgx](#). pages 264–278, 05 2018. doi:10.1109/SP.2018.00025.
- [22] Raluca Popa, Catherine Redfield, Nikolai Zeldovich, and Hari Balakrishnan. [Cryptdb: Processing queries on an encrypted database](#). *Communications of The ACM - CACM*, 55, 09 2012. doi:10.1145/2330667.2330691.
- [23] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. volume 5, pages 223–238, 05 1999. ISBN: 978-3-540-65889-4.
- [24] Microsoft. [Always encrypted](#). Online, Acedido em 2020.
- [25] Microsoft. [Always encrypted cryptography](#). Online, Acedido em 2020.
- [26] AWS. [Protecting data using client-side encryption](#). Online, Acedido em 2020.