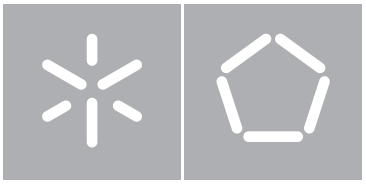
Universidade do Minho
Escola de Engenharia

Mário Jorge Barros Pinto

Computação segura de SQL

Outubro de 2014



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Mário Jorge Barros Pinto

Computação segura de SQL

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor Rui Carlos Mendes Oliveira
Professor Doutor Manuel Bernardo Martins Barbosa

Outubro de 2014

Universidade do Minho

REQUERIMENTO DE REALIZAÇÃO DE PROVAS DE DISSERTAÇÃO / TRABALHO DE PROJETO

MESTRADO EM ENGENHARIA INFORMÁTICA

Não é permitida a entrega de documentos manuscritos

ALUNO Nome *(Completo)* Mário Jorge Barros Pinto

Nº PG22788 **ANO LECTIVO:** 2013 / 2014

E-mail: mariobpinto@gmail.com

TÍTULO DISSERTAÇÃO/TRABALHO DE PROJETO:

Título em PT : Computação segura de SQL

Título em EN : Secure computation of SQL

DATA DE ENTREGA: 31 / 10 / 2014

ASSINATURAS

Orientador:

Co-orientador: MBB

Aluno Mário Jorge Barros Pinto

Declaro que este trabalho é original e foi especialmente elaborado para a conclusão da UC ☑

Acesso embargado entre 1 e 3 anos ☐

Anexo justificação do autor e do(s) seu(s) supervisor(es) para embargo superior a 3 anos (contados a partir da data da defesa) ☐

Este Requerimento será entregue ao Diretor de Curso e devem ser atendidas as normas específicas da respetiva Direção de Curso.

Observações:

Este trabalho foi suportado pela European Union Seventh Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE).

Abstract

Due to the emerging of cloud-based services, people have been, gradually, changing the way they store their data. The possibility to store data on a remote location, accessible from anywhere, at any time, with the option to, easily and at any time, adjust the services depending on their needs, are characteristics that are greatly appreciated by the users. However, recent media scandals, regarding governmental surveillance programs, arose some trust issues regarding these cloud-based services. The usage of these services, have been made with greater care, because no one wants their data to be visible to those services, nor to unauthorized third parties. The need to develop platforms which are capable of storing and computing data in a privately and securely manner arises.

Hoping to contribute to this need, this dissertation presents a secure storage system, using secure multi-party computation techniques, able to store and process secret shared data, preserving the ability to query the system using SQL.

Trying to bridge the performance gap between this secure system and the common ones, we introduce secret sharing algorithms, based on those used in the secure computation platform Sharemind, offering tradeoffs between security and performance. With those algorithms, this system provides the option to, depending on the client's needs, adjust the security and performance configurations.

Resumo

Com o surgimento dos serviços baseados na *cloud*, as pessoas têm, gradualmente, alterado a forma como armazenam os seus dados. A possibilidade de armazenar informação num local remoto, acessível a partir de qualquer lugar, em qualquer altura, e ter a possibilidade de, facilmente, a qualquer momento, ajustar os serviços consoante as necessidades, são características muito apreciadas pelos utilizadores. Porém, com os recentes escândalos mediáticos, relativamente aos programas de vigilância governamentais, a confiança nestes serviços tem diminuído. A utilização destes serviços, passou a ser feita com relutância, pois, geralmente, ninguém deseja que os seus dados, armazenados ou processados por estes serviços, sejam visíveis a estes ou a terceiros não autorizados. Manifesta-se, assim, a necessidade de desenvolver plataformas capazes de armazenar e computar dados de forma segura e privada.

No âmbito de contribuir para esta necessidade, esta dissertação apresenta um sistema de armazenamento de dados seguro, recorrendo a técnicas de *Secure Multi-Party Computation*, capaz de armazenar e processar dados, cifrados com algoritmos de *secret sharing* e que, tal como os sistemas de dados comuns, pode ser consultado usando a linguagem de *query SQL*.

De forma a tentar aproximar o desempenho deste sistema seguro, ao dos sistemas de armazenamento de dados comuns, apresentamos algoritmos de *secret sharing*, baseados em algoritmos apresentados na plataforma de computação segura *Sharemind*, que oferecem *tradeoffs* entre segurança e desempenho. Com isto, este sistema de dados seguro oferece a possibilidade de ajustar, mediante as necessidades do utilizador, as propriedades de desempenho e segurança.

Agradecimentos

Ao meu orientador, Professor Rui Carlos Oliveira, que ao longo de todo o projeto me motivou, guiou e sempre demonstrou um grande entusiasmo pelo que estávamos a tentar alcançar, dando-me assim a força necessária para superar as adversidades que foram surgindo.

Ao meu co-orientador, Professor Manuel Bernardo Barbosa, que sempre se mostrou disponível para me ajudar, com toda a sua experiência e conhecimento, a entender desde os conceitos mais simples aos mais complexos, motivando-me ao longo do percurso.

Aos meus colegas do Grupo de Sistemas Distribuídos que sempre se mostraram disponíveis e prestáveis para qualquer coisa que precisasse. Um especial obrigado ao Ricardo Vilaça que, desde o início, e com toda a paciência do mundo, me acompanhou e ajudou a ultrapassar grande parte dos problemas que foram surgindo.

Ao Bernardo Portela que sempre demonstrou um grande interesse no meu trabalho e total disponibilidade para me ajudar, vezes sem conta, a tomar as melhores decisões.

Aos meus pais que desde sempre acreditaram em mim e me deram esta oportunidade, mesmo quando atravessavam períodos mais frágeis e complicados. Nada disto seria possível sem eles.

Aos meus amigos, que nunca deixaram de me apoiar, mesmo nas alturas de maior stress e desespero. Em especial à Paula Lisboa, que esteve sempre por perto e que releu este documento vezes sem conta, à procura de erros, contribuindo para uma dissertação escrita com uma maior qualidade.

Por último, quero agradecer todos aqueles que de alguma forma tenham contribuído para o desenvolvimento desta dissertação, mesmo não estando aqui mencionados.

Conteúdo

1 Introdução **1**

1.1 Contextualização 1
1.2 Motivação 2
1.3 Objetivos e contribuições 2
1.4 Estrutura do documento 3

2 Trabalho Relacionado **5**

2.1 Computação segura 5
2.1.1 Conceito 6
2.1.2 Técnicas de Secret Sharing 6
2.1.3 Sharemind 8
2.2 Segurança no *Sharemind* 11
2.2.1 A *framework* UC 11
2.2.2 Adaptação para o *Sharemind* 13
2.2.3 Prova de segurança para a multiplicação 15
2.3 Sistemas de armazenamento de dados 19
2.3.1 Sistemas de dados relacionais 19
2.3.2 Sistemas de dados não-relacionais 20
2.3.3 Sistemas de armazenamento de dados seguro 21

3 Algoritmos com segurança ajustável **24**

3.1 Determinar a relação de igualdade entre dois valores 24
3.1.1 Equal(u,v) - Versão *Sharemind* 25

3.1.2 Equal(u,v) - Versão A 26
3.1.3 Equal(u,v) - Versão B 28
3.2 Determinar a relação de desigualdade entre dois valores 31
3.2.1 GreaterThan(u,v) - Versão *Sharemind* 31
3.2.2 GreaterThan(u,v) - Versão A 32
3.2.3 GreaterThan(u,v) - Versão B 33

4 Validação experimental **35**
4.1 Arquitetura do sistema 35
4.2 *Queries* suportadas 37
4.3 Especificação do sistema 39
4.4 Implementação 39
4.4.1 Implementação em bases de dados relacionais 40
4.4.2 Implementação em bases de dados não-relacionais 40
4.5 Análise de desempenho 41
4.5.1 Considerações gerais 41
4.5.2 Ambiente e configuração 42
4.5.3 Tempos de execução 43

5 Conclusão **49**
5.1 Trabalho futuro 50

A Algoritmos de *Secret Sharing* **55**

Lista de Figuras

2.1 *Secret sharing* numa base de dados relacional 9
2.2 Modelo do *Sharemind* 10
2.3 Esquematização do mundo real 14
2.4 Esquematização do mundo ideal 14
2.5 Funcionalidade Ideal $\mathcal{F}$ 16
2.6 Encriptação em camadas do *CryptDB* 23

4.1 Arquitetura do sistema em *Derby* 36
4.2 Arquitetura do sistema em *HBase* 37
4.3 Tempo de execução de uma *query* que avalia um predicado de igualdade em 500000 linhas 44
4.4 Tempo de execução de uma *query* que avalia um predicado de desigualdade em 500000 linhas 47

Lista de Tabelas

4.1 Exemplo da estrutura da tabela da base de dados 42
4.2 Tempo de execução de uma *query* que avalia um predicado de igualdade em 500000 linhas 43
4.3 Tempo de execução de uma *query* que avalia um predicado de desigualdade em 500000 linhas 46

Lista de Algoritmos

1 Algoritmos de *secret sharing* aditivo 8
2 Protocolo $[\![u]\!] \leftarrow \text{Share}(u)$ para gerar *shares* de u 10
3 Protocolo $[\![w]\!] \leftarrow \text{Mult}([\![u]\!],[\![v]\!])$ para multiplicar dois valores partilhados em $\mathbb{Z}_{2^n}$ 15

4 Protocolo $[\![w]\!] \leftarrow \text{Equal}([\![u]\!], [\![v]\!])$ para avaliar o predicado de igualdade. 25
5 Protocolo $[\![w]\!] \leftarrow \text{EqualA}([\![u]\!], [\![v]\!])$ para avaliar o predicado de igualdade. . . . 27
6 Protocolo $[\![w]\!] \leftarrow \text{EqualB}([\![u]\!], [\![v]\!])$ para avaliar o predicado de igualdade. . . . 29
7 Protocolo $[\![w]\!] \leftarrow \text{GreaterThan}([\![u]\!], [\![v]\!])$ para avaliar o predicado de maior ou igual. 32
8 Protocolo $[\![w]\!] \leftarrow \text{GreaterThanA}([\![u]\!], [\![v]\!])$ para avaliar o predicado de maior ou igual. 33
9 Protocolo $[\![w]\!] \leftarrow \text{GreaterThanB}([\![u]\!], [\![v]\!])$ para avaliar o predicado de maior ou igual. 34

10 Protocolo $[\![u]\!]$ $\text{Deshare}(u_1, u_2, u_3)$ para recuperar u a partir das suas *shares* . . . 55
11 Protocolo $[\![w]\!] \leftarrow \text{Reshare}([\![u]\!])$ para *resharing* 55
12 Protocolo $[\![u'\!]\!] \leftarrow \text{ReshareToTwo}([\![u]\!])$ para fazer reshare a um valor $[\![u]\!]$ entre $\mathcal{P}_2$ e $\mathcal{P}_3$ 56
13 $\overline{[\![p'\!]\!]} \leftarrow \text{PrefixOR}(\overline{[\![p]\!]})$ 56
14 Protocolo $[\![s]\!] \leftarrow \text{MSNZB}(\overline{[\![u]\!]})$ para determinar a posição do bit não-zero mais significativo 57
15 Protocolo $[\![\lambda]\!] \leftarrow \text{Overflow}([\![u'\!]\!])$ para obter o bit de overflow $[\![\lambda]\!]$ para $[\![u'\!]\!]$ se a *share* $u'_1 = 0$ 57
16 Protocolo $[\![v]\!] \leftarrow \text{ShareConv}([\![u]\!])$ para converter uma *share* $[\![u]\!] \in \mathbb{Z}_2$ para $[\![v]\!] \in \mathbb{Z}_{2^n}$ 58

17 Protocolo $[\![w]\!] \leftarrow \text{ShiftR}([\![u]\!], \text{p})$ para avaliar o shift à direita 59
18 Protocolo $[\![w]\!] \leftarrow \text{BitConj}(\overline{[\![b]\!]})$ para determinar a conjunção dos bits de b . . . 59

Capítulo 1

Introdução

1.1. Contextualização

Para a computação segura sobre dados privados são necessárias operações que, idealmente, deverão ser integradas nos próprios sistemas de gestão de dados.

Existem atualmente implementações destas operações sobre sistemas de dados relacionais tradicionais embora não integradas no sistema de gestão de base de dados.

Com a generalização do paradigma de *cloud computing*, a proliferação de aplicações à escala da Internet, e o crescimento sem precedentes do volume de dados associados a ambientes de *BigData* e *Internet of Things* o tradicional modelo de bases de dados relacionais tem-se revelado limitado, sendo a solução natural a adoção de bases de dados não-relacionais *NoSQL*. Estes sistemas permitem gerir dados estruturados, semi-estruturados e não-estruturados de forma mais eficiente e escalável. No entanto, para bases de dados *NoSQL* não há na literatura nem em propostas comerciais ainda quaisquer mecanismos de computação segura.

Como parte do objectivo último de desenvolver uma plataforma de *Secure Storage as a Service* capaz de armazenar e processar dados com total privacidade e segurança em ambiente de *cloud* do projeto EU-FP7 *PRACTICE*[1], esta dissertação visa contribuir com a investigação e o desenvolvimento de componentes de coprocessamento para integração de computação segura

[1] http://practice-project.eu/

em sistemas de bases de dados relacionais e *NoSQL*.

1.2. Motivação

Nos últimos anos, tem-se verificado um crescente interesse nos serviços baseados na *cloud*. Este tipo de serviços abre portas a um vasto leque de possibilidades, entre elas o *outsourcing* de computação e armazenamento, com as ótimas características de disponibilidade, fiabilidade e escalabilidade que os serviços *cloud* tipicamente oferecem e relações custo-benefício imbatíveis.

Mais recentemente, com a revelação de vários programas clandestinos de vigilância de agências governamentais, a segurança e privacidade na *web* tem ganho uma nova importância e mais atenção por parte da população em geral. Neste novo cenário, os serviços *cloud*, mais do que nunca, levantam questões ao nível da segurança e privacidade dos dados armazenados/-processados, pois não é, de forma alguma, desejável que dados confidenciais e/ou sensíveis estejam ao alcance de terceiros não autorizados.

É, por isso, importante desenvolver plataformas de computação e armazenamento seguro, em que os dados permaneçam confidenciais aos hospedeiros, outros utilizadores do serviço e curiosos, preservando a possibilidade de realizar, de forma transparente, computações sobre os dados. Essencialmente, desenvolver plataformas de computação segura que ofereçam fortes garantias de segurança e privacidade sobre os dados.

Paralelamente, com o crescimento exponencial de informação gerada diariamente, torna-se interessante adaptar este tipo de plataformas de computação segura, de modo a suportarem bases de dados não-relacionais. Desta forma, podemos juntar as propriedades de segurança que estas plataformas oferecem, com a excelente capacidade de escalabilidade deste paradigma de bases de dados.

1.3. Objetivos e contribuições

Apresentamos, nesta dissertação, um protótipo de um sistema de computação e armazenamento seguro, baseado em conceitos de *Secure Multi-Party Computation* e técnicas de *secret sharing*.

Num primeiro passo, desenvolvemos o sistema integrado numa bases de dados relacional, alterando-a internamente de forma a suportare operações sobre dados *secret shared*.

Numa segunda fase, este sistema de dados relacional foi transposto para o modelo não-relacional. Para tal, foi desenvolvido um componente de coprocessamento para um sistema de dados não-relacional, que usa conceitos de computação segura, baseados em técnicas de *secret sharing*, e que tira partido das excelentes características de desempenho e escalabilidade dos sistemas de dados não-relacionais. Tendo como base outro projeto [25] do High-Assurance Software Laboratory (INESC TEC e Universidade do Minho) [2], acrescenta-se a possibilidade de executar *queries* SQL sobre o sistema de dados não-relacional, algo que, tipicamente, não é suportado por este tipo de sistema de dados. Desta forma, o sistema permite a execução de um subconjunto de *queries* SQL sobre dados *secret shared*, armazenados em sistemas de dados não-relacionais.

Por último, e de particular relevância prática, são discutidas configurações aplicáveis aos algoritmos de *secret sharing* usados no sistema desenvolvido que permitem ajustar, mediante as necessidades, o equilíbrio entre privacidade e desempenho, objetivos invariavelmente em confronto.

1.4. Estrutura do documento

Para além deste capítulo introdutório, esta dissertação está estruturada em mais quatro capítulos e um anexo:

Capítulo 2: É exposto o trabalho relacionado, focado nos conceitos armazenamento e computação segura, as noções de *Secure Multi-Party Computation* e as técnicas *secret sharing*, essenciais para o entendimento deste trabalho.

Capítulo 3: São apresentados os algoritmos de *secret sharing* utilizados no *Sharemind*, juntamente com versões alternativas, desenvolvidas para esta dissertação, que visam oferecer *tradeoffs* entre segurança e desempenho.

[2] http://haslab.pt/

Capítulo 4: É discutida a implementação do sistema proposto, quer no modelo relacional, quer no não-relacional, e é apresentada, e discutida, a avaliação de desempenho do sistema.

Capítulo 5: Neste capítulo concluímos o trabalho apresentado nesta dissertação, discutindo possíveis direções para trabalho futuro.

Anexo A: Listagem dos algoritmos de *secret sharing* utilizados no sistema de dados seguro apresentado.

Capítulo 2

Trabalho Relacionado

2.1. Computação segura

A informação processada por empresas, organizações governamentais ou indivíduos é frequentemente confidencial, o que deve ser respeitado pelas entidades que processam essa informação. Isto é relativamente fácil de controlar quando a informação é processada localmente, mas bastante mais difícil de garantir quando o processamento é providenciado por uma entidade externa. Os serviços *cloud* oferecem grandes benefícios tanto em termos financeiros como em termos de fácil acesso e disponibilidade de dados e serviços. Organizações e indivíduos optam, por isso, e cada vez mais, por fazer *outsourcing* dos seus dados para uma *cloud* onde, uma entidade, que pode ou não ser de confiança, trata da computação e do armazenamento dos dados. Esta é uma das grandes preocupações dos serviços *cloud*: o entrave em conseguir que o cliente confie nas medidas de segurança disponibilizadas pelo serviço.

Técnicas comuns de computação não podem ser aplicadas sobre dados cifrados, o que obriga a que os dados sejam decifrados antes de serem processados. Uma solução para este problema pode passar por primitivas criptográficas de computação segura. Estas primitivas permitem a computação distribuída de funções arbitrárias sobre dados confidenciais, escondendo qualquer tipo de informação sobre os mesmos. Dito de outra forma, estas primitivas suportam computações sobre dados cifrados. Contudo, primitivas deste tipo são consideravelmente mais

complexas do que as primitivas de computação comuns.

2.1.1. Conceito

A computação segura é um ramo da criptografia que visa permitir a realização de um determinado conjunto de computações sobre dados cifrados, preservando a confidencialidade dos mesmos. Por sua vez, o armazenamento seguro é um conjunto de técnicas que possibilitam o armazenamento de dados de forma a que estes fiquem disponíveis para quem possui permissão para eles e indisponíveis para os restantes. Numa altura em que os ambientes *cloud* são cada vez mais populares, a computação e o armazenamento seguros ganham uma importância redobrada. Este paradigma permite armazenar e, eventualmente, processar dados num servidor na *cloud* sem que a confiabilidade do servidor, ou a privacidade dos dados, sejam uma preocupação.

Quando a computação segura acontece num ambiente distribuído, isto é, em que o objetivo é a computação de dados conjuntamente com outras entidades de forma distribuída, esta é conhecida como *Secure Multi-Party Computation* (SMPC). Esta configuração permite que várias entidades consigam realizar computações sobre a sua parte de um segredo sem a revelar aos restantes participantes e sem ter de recorrer a uma *trusted third-party*. Eliminar por completo a necessidade de recorrer a uma *trusted third-party* é um dos objetivos principais dos protocolos de *Secure Multi-Party Computation*.

O exemplo canónico de *Secure Multi-Party Computation* é o problema dos milionários proposto por Yao [26] em 1982, em que dois milionários pretendem descobrir quem é o mais rico sem revelar ao outro, ou a terceiros, a sua riqueza nem qualquer outro tipo de informação adicional.

2.1.2. Técnicas de Secret Sharing

O *secret sharing*, ou partilha de segredos, é uma técnica proposta em 1979 por Adi Shamir e George Blakley. Esta consiste na distribuição de um segredo entre n participantes em que, cada um, recebe uma parte (*share*) do segredo. Cada *share*, por si só, não revela informação sobre o segredo original e, para reconstruir o segredo, é necessário reunir um determinado número

mínimo de *shares*. Esta técnica é ideal para proteger informação confidencial e de elevada importância (e.g., chaves secretas, códigos bancários, códigos militares).

Técnicas criptográficas tradicionais, de forma geral, não conseguem garantir, em simultâneo, máxima disponibilidade e confidencialidade. Obrigam a escolher entre manter uma cópia do segredo num único local ou manter várias cópias distribuídas por várias entidades. No primeiro caso, atinge-se uma maior confidencialidade pois só um participante conhece o segredo. No segundo, existe uma maior disponibilidade mas existem mais oportunidades para o segredo cair em mãos alheias, comprometendo assim a confidencialidade. Ao contrário das técnicas criptográficas tradicionais, o *secret sharing* permite atingir, simultaneamente, arbitrários níveis de confidencialidade e disponibilidade.

Existem vários esquemas de *secret sharing*, sendo o mais conhecido o de Shamir [24] que permite que o segredo seja recuperado utilizando um número k das n *shares* do segredo. Esta propriedade é útil pois nem sempre é prático conseguir reunir todos os participantes de forma a recuperar o segredo. A ideia por trás do esquema de Shamir assenta na interpolação polinomial, dados k pontos, num plano bidimensional.

Outro esquema de *secret sharing* é aquele usado pelo *Sharemind* [3], uma plataforma de computação segura, que é, aditivamente, homomórfica para inteiros (alg 1). A propriedade homomórfica deste esquema permite que se possam fazer certas computações sobre os dados partilhados com *secret sharing*. Cada segredo, S, é dividido em três *shares* e a reconstrução do segredo original implica o acesso às três. Para gerar as *shares* são gerados três inteiros (s_1, s_2 e s_3). Os dois primeiros, s_1 e s_2, são gerados com ajuda de um gerador de números pseudo-aleatórios e o último, s_3, é a diferença entre o segredo original e os dois inteiros gerados aleatoriamente módulo $Z_{2^{32}}$ (alg 2).

Algoritmo 1: Algoritmos de *secret sharing* aditivo

Dados: Valor secreto s

Resultado: Valores partilhados $s_1, s_2, ..., s_n$ tal que $s \equiv (s_1 + s_2 + ... + s_n) \mod \mathbb{Z}_{2^m}$

1 Gera aleatoriamente $s_1 \leftarrow \mathbb{Z}_{2^m}$

2 Gera aleatoriamente $s_2 \leftarrow \mathbb{Z}_{2^m}$

3 ...

4 Gera aleatoriamente $s_{n-1} \leftarrow \mathbb{Z}_{2^m}$

5 Calcula $s_n = (s - s_1 - s_2 - ... - s_{n_1}) \mod \mathbb{Z}_{2^m}$

6 Retorna $[\![s]\!]$

Desta forma, a reconstrução de S, no caso do *Sharemind*, é, como podemos ver na Fórmula 2.1, a soma das três *shares* computadas.

$$S \equiv (s_1 + s_2 + s_3) \mod 2^{32} \tag{2.1}$$

Na Figura 2.1 podemos ver como é aplicado o *secret sharing* numa base de dados relacional, na plataforma *Sharemind* com três participantes. Na secção 2.1.3 é explicado com mais detalhe o funcionamento deste esquema de *secret sharing* e do *Sharemind*.

Existem outros esquemas de *secret sharing* baseados em diversas técnicas tais como o teorema chinês dos restos ou a intersecção de hiperplanos não-paralelos [2], que não são homomórficos para as operações de adição ou multiplicação.

O sistema proposto no capítulo 4 funciona sobre dados partilhados na forma *secret shared*.

2.1.3. Sharemind

O *Sharemind* [3] é um sistema de computação seguro, capaz de processar um input sem comprometer a sua privacidade, isto é, consegue processar dados sem os ver. Funciona como um sistema de *multi-party computation* seguro, divido em três partes: as entidades fornecedoras de dados (*input parties*), as entidades de computação (*computing parties*) e as entidades que pedem computações designadas (*result parties*). A relação entre as três partes pode ser visualizada na Figura 2.2. Podemos ter um número arbitrário (sempre superior a zero) de *input*

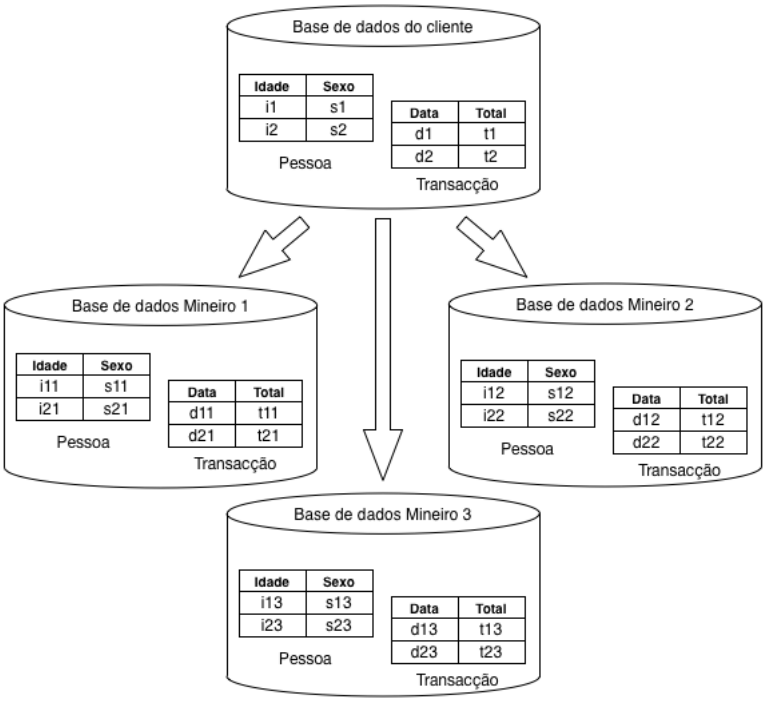


Figura 2.1.: *Secret sharing* numa base de dados relacional

parties e de *result parties* e, por definição, três *computing parties*. O número de *computing parties* foi definido com base no menor número necessário para atingir segurança contra adversários passivos, em que menos de metade dos participantes é corrupto. Esta plataforma é particularmente útil para executar algoritmos de mineração ou de agregação sobre dados partilhados.

O modo de funcionamento do *Sharemind* é o seguinte: as *input parties* aplicam *secret sharing* aos seus dados (alg. 2) e enviam uma *share* a cada uma das três *computing parties* através de canais de comunicação seguros previamente estabelecidos. Dado que todas as *input parties* podem não estar *online* simultaneamente, é necessário um mecanismo de armazenamento seguro para armazenar os *inputs*, para que estes possam ser processados mais tarde. Para tal, o *Sharemind* usa uma técnica de *secret sharing* aditivamente homomórfico, como foi referido na secção 2.1.2. Cada segredo é divido em três *shares* que, por si só, não revelam qualquer tipo de informação, garantindo assim uma segurança incondicional (*information-theoretic*

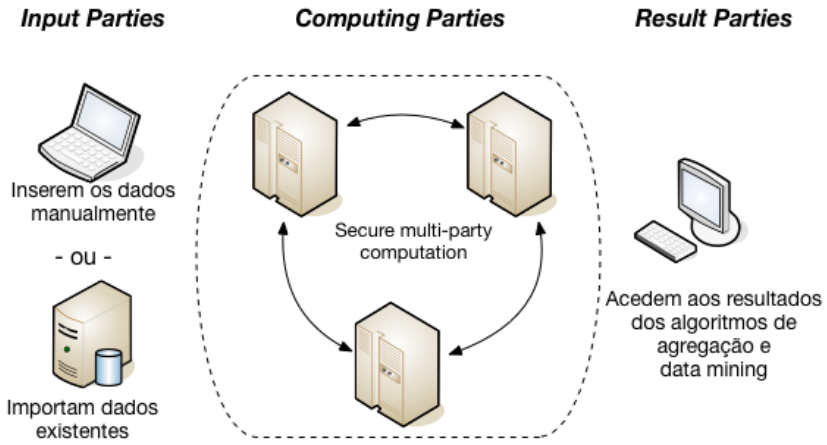


Figura 2.2.: Modelo do *Sharemind*

security)[1]. Após todos os dados terem sido enviados às *computing parties*, as *result parties* podem pedir computações sobre estes dados. Para tal, cada *result party* pode invocar, sobre os dados armazenados, a execução de algoritmos previamente inseridos na plataforma *Sharemind*. Cada *computing party* executa os algoritmos sobre as suas *shares* e envia os resultados obtidos para cada *result party* que pediu a computação. Por fim, cada *result party* junta as *shares* que recebeu de cada *computing party* de forma a construir o resultado final.

Algoritmo 2: Protocolo $[\![u]\!] \leftarrow \text{Share}(u)$ para gerar *shares* de u

Dados: Valor $u \in \mathbb{Z}_{2^n}$

Resultado: Valores u_1, u_2 e u_3 tal que $u \equiv (u_1 + u_2 + u_3) \mod \mathbb{Z}_{2^n}$

1 Gera aleatoriamente $u_1 \leftarrow \mathbb{Z}_{2^n}$

2 Gera aleatoriamente $u_2 \leftarrow \mathbb{Z}_{2^n}$

3 Calcula $(u_3 = u - u_1 - u_2) \mod \mathbb{Z}_{2^n}$

4 Retorna $[\![u]\!]$

O modelo original do *Sharemind* permite realizar *multi-party computation* sobre inteiros [4]. A propriedade homomórfica aditiva que possui permite realizar de forma trivial adições e,

[1] Não existe nenhum adversário, mesmo com poder computacional ilimitado, que consiga quebrar este tipo de segurança. Só com uma *share*, não exste informação suficiente para reconstruir o valor original.

de forma composta, multiplicações. A composição destas duas operações basilares permite derivar outras operações, tais como divisões ou comparações.

Mais recentemente, foram apresentados por Kamm e Willemson [14] métodos para realizar operações aritméticas sobre números de vírgula flutuante. Estes métodos foram apresentados como parte de um mecanismo de detecção de colisões entre satélites, usando a plataforma *Sharemind*.

O sistema proposto, apresentado na secção 4, usa algumas destas técnicas e conceitos apresentados no *Sharemind*.

2.2. Segurança no *Sharemind*

Explicamos, nesta secção, a noção de segurança nos protocolos do *Sharemind*. Os algoritmos do *Sharemind* foram provados seguros em [7] , utilizando o modelo de *Reactive Simulability*[1].

As provas de segurança que apresentamos nesta dissertação seguem uma adaptação do modelo de *universally composable security* (UC) de Canetti [8], por acharmos que este modelo, sendo mais usado e estandardizado, permite apresentar provas mais claras.

Como exemplo, apresentamos a prova de segurança para o algoritmo de multiplicação (alg.3), descrevendo, primeiramente, as entidades do modelo UC e as noções de segurança necessárias à compreensão das provas.

2.2.1. A *framework* UC

A *framework* UC é um sistema que pretende estabelecer e analisar a segurança de protocolos criptográficos, em ambientes de rede inseguros, e em que os protocolos interagem com um número arbitrário de outros protocolos.

Comecemos por definir, tendo como base [23, 8, 12, 9], as entidades da *framework* UC:

Funcionalidades Ideais ($\mathcal{F}$) representam tarefas criptográficas desejáveis; no modelo UC, uma funcionalidade ideal representa uma entidade de confiança que recebe *inputs* das *input parties* $\mathcal{P}_0$,$\mathcal{P}_1$,...,$\mathcal{P}_n$, executa uma determinada tarefa computacional sobre esses *inputs*, e retorna os *outputs* corretos às *output parties*. Uma funcionalidade ideal não pode ser corrompida, sendo

sempre honesta. Tal funcionalidade não pode existir no mundo real, dado as assunções sobre a sua incorruptibilidade não serem realistas.

Protocolos (π) são construções reais que tentam emular a funcionalidade ideal $\mathcal{F}$. Tipicamente, uma *party* honesta interage com a funcionalidade $\mathcal{F}$ executando o protocolo π; *parties* corruptas interagem com $\mathcal{F}$ executando código arbitrário.

Parties ($\mathcal{P}_i$) são os participantes que executam localmente as diferentes partes do protocolo. Estas podem ser concretas ou *dummy parties* [2], sendo que as concretas podem ser honestas ou corruptas.

Vista (V_i) representa todos os valores que a *party* $\mathcal{P}_i$ vê durante a execução do protocolo. Isto inclui os *inputs* recebidos, os valores gerados e valores recebidos de outras *parties*.

Mundo real configuração onde as *parties* concretas executam um protocolo π, eventualmente ,na presença de um adversário $\mathcal{A}$ que pode controlar a comunicação de, e para, as *parties* corrompidas.

Mundo ideal configuração onde, no lugar das *parties*, existe uma funcionalidade ideal $\mathcal{F}$ que executa, em segurança, um protocolo π.

Ambiente ($\mathcal{Z}$) modela toda a atividade na rede, externa às *parties* que executam o protocolo π ou que, de outra forma, interagem com $\mathcal{Z}$. O ambiente interage com todas as *parties* e adversários.

Adversários ($\mathcal{A}$, $\mathcal{S}$) tentam extrair informações secretas das *parties* honestas e/ou tentam manipular a execução do protocolo, corrompendo uma ou mais *parties*. A corrupção de uma *party* honesta é modelada, de forma a permitir ao adversário controlo total sobre a interação da *party* corrompida com a funcionalidade ideal $\mathcal{F}$. O adversário também comunica com o ambiente $\mathcal{Z}$.

[2] *Parties* construídas para que se assemelhem e substituam uma *party* concreta

Descrevemos agora a noção de segurança, tendo como base a dualidade entre o mundo real e o mundo ideal.

> "A protocol is secure for some task if it "emulates"an "ideal setting"where the parties hand their inputs to a "trusted party", who locally computes the desired outputs and hands them back to the parties." (Goldreich-Micali-Wigderson87)

Para um dado protocolo π, um adversário $\mathcal{A}$, uma funcionalidade $\mathcal{F}$ e um parâmetro de segurança k:

Definição 1. *Um protocolo π realiza com segurança $\mathcal{F}$ em relação a $\mathcal{A}$, se para qualquer ambiente adversário $\mathcal{Z}$, existe um simulador $\mathcal{S}$ com complexidade polinomial em $\mathcal{Z}$ tal que, para qualquer* input *z, temos um valor negligenciável k tal que :*

$$Vista(REAL_{\pi,\mathcal{A}}(k,z)) \approx Vista(IDEAL_{\mathcal{F},\mathcal{S},\mathcal{A}}(k,z))$$

Quando temos vistas indistinguíveis para qualquer *input* e *output*, $\mathcal{Z}$ não consegue distinguir, com uma probabilidade não negligenciável, se está a interagir com o protocolo do mundo real (fig. 2.3) ou com a funcionalidade do mundo ideal (fig. 2.4).

2.2.2. Adaptação para o *Sharemind*

O *Sharemind* foi desenvolvido com um modelo de segurança mais fraco em mente, o qual passamos a descrever. No modelo do *Sharemind*, existem três entidades computacionais distintas $\mathcal{P}_1$, $\mathcal{P}_2$ e $\mathcal{P}_3$ e existe, no mundo real, um protocolo p que realiza, em segurança, a funcionalidade de transmissão de mensagens [11]. Desta forma, assumimos que existem canais de comunicação seguros entre as entidades computacionais.

A segurança é provada no modelo passivo (honesto-mas-curioso), onde se assume que, no máximo, podemos ter $1/3$ dos adversários corrompidos. Como o número de *computing parties* no *Sharemind* é fixado em três, assumimos que o adversário $\mathcal{A}$ tem poder para corromper, no máximo, uma das três entidades computacionais, antes da execução do protocolo. Após

o início deste, o adversário recebe quer os *inputs* da entidade corrompida, quer todas as mensagens transmitidas de, e para, a entidade corrompida (curiosidade) mas não tem qualquer influência sobre os *outputs*. Os *outputs* continuam a ser gerados conforme indicado no protocolo (honestidade). É absolutamente necessário que pelo menos duas *parties* se mantenham honestas, no decorrer do protocolo, caso contrário a segurança estabelecida deixa de ser válida.

Para provar que um protocolo, π, satisfaz a nossa adaptação ao basta mostrar que este é perfeitamente simulável. Para provar a simulabilidade, consideramos as vistas (*views*) de cada *party* $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$, e provamos que são independentes das *shares* dos *inputs* das outras *computing parties*, provando a existência de um simulador $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$, para cada uma delas.

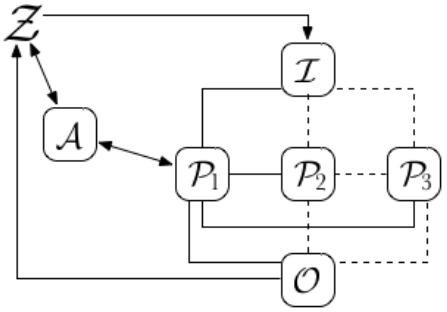


Figura 2.3.: Esquematização do mundo real

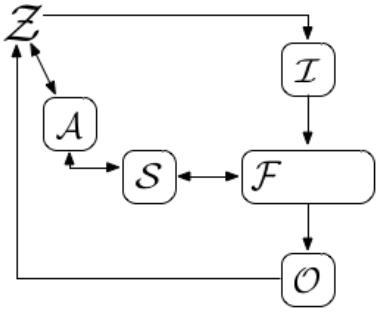


Figura 2.4.: Esquematização do mundo ideal

2.2.3. Prova de segurança para a multiplicação

Começamos por apresentar o protocolo de multiplicação. Neste protocolo, cada *computing party* recebe uma *share* de um valor $[\![u]\!]$ e outra de um valor $[\![v]\!]$ e, após a execução do protocolo, retorna uma *share* de um valor $[\![w]\!]$ tal que, w corresponde a uv. Podemos ver, no algoritmo 3, a execução passo-a-passo do protocolo.

Algoritmo 3: Protocolo $[\![w]\!] \leftarrow$ Mult($[\![u]\!]$,$[\![v]\!]$) para multiplicar dois valores partilhados em $\mathbb{Z}_{2^n}$

Dados: Valores partilhados $[\![u]\!]$ e $[\![v]\!]$ fornecidos pela *input party*

Resultado: Valor $[\![w']\!]$ partilhado tal que w'=uv , enviado à *output party*

1 $[\![u']\!] \leftarrow$ Reshare($[\![u]\!]$)
2 $[\![v']\!] \leftarrow$ Reshare($[\![v]\!]$)
3 $\mathcal{P}_1$ envia u'_1 e v'_1 a $\mathcal{P}_2$
4 $\mathcal{P}_2$ envia u'_2 e v'_2 a $\mathcal{P}_3$
5 $\mathcal{P}_3$ envia u'_3 e v'_3 a $\mathcal{P}_1$
6 $\mathcal{P}_1$ calcula $w_1 \leftarrow u'_1v'_1 + u'_1v'_3 - u'_3v'_1$
7 $\mathcal{P}_2$ calcula $w_2 \leftarrow u'_2v'_2 + u'_2v'_1 - u'_1v'_2$
8 $\mathcal{P}_3$ calcula $w_3 \leftarrow u'_3v'_3 + u'_3v'_2 - u'_2v'_3$
9 Retorna $[\![w']\!] \leftarrow$ Reshare($[\![w]\!]$)

Necessitamos, agora, de provar que o protocolo π realiza, com segurança, a funcionalidade ideal $\mathcal{F}$. Começamos por definir a funcionalidade ideal.

Funcionalidade Ideal

A seguinte funcionalidade ideal recebe, como *inputs*, os valores u e v, determina $w = uv$ e, por fim, retorna w. A funcionalidade ideal é retratada da seguinte forma:

- $\mathcal{F}$ recebe os *inputs* u e v da *input party*;
- $\mathcal{F}$ determina w computando $w = uv$;
- $\mathcal{F}$ retorna w à *output party*;

A figura 2.5 mostra a esquematização da funcionalidade ideal $\mathcal{F}$, onde esta recebe um *input*, executa o protocolo ideal e devolve o *output* correto.

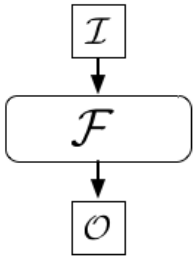


Figura 2.5.: Funcionalidade Ideal $\mathcal{F}$

Para provar a segurança do algoritmo 3, notamos que este é simétrico para as *parties* $\mathcal{P}_1$ e $\mathcal{P}_2$. A execução do protocolo é igual para as duas e, os seus valores são todos gerados da mesma forma. No caso de $\mathcal{P}_3$, a única diferença reside na forma como as suas *shares*, u_3, v_3, do *input* são geradas. As *shares* u_1, u_2 e v_1, v_2 são geradas de forma aleatória, como podemos ver no algoritmo 2. Já u_3 e v_3 resultam das seguintes computações:

$$u_3 = u - u_1 - u_2$$
$$v_3 = v - v_1 - v_2$$

Mostremos agora que estas duas *shares* são, igualmente, uniformemente distribuídas, usando o teorema e prova apresentados em [3]:

Teorema 1. *Para cada valor secreto $s \in \mathbb{Z}_{2^m}$, qualquer subconjunto de $n-1$* shares *de s é uniformemente distribuído e, para qualquer dois valores secretos $u, v \in \mathbb{Z}_{2^m}$, as suas* shares *são indistinguíveis para* parties*, em conluio, na posse de até $n-1$* shares*.*

Prova 1. *De acordo com o algoritmo 2, as* shares *$s_1, s_2, \ldots, s_{n-1}$ são uniformemente geradas em $\mathbb{Z}_{2^m}$. Queremos agora mostrar que as* shares *$s_2, s_3, \ldots, s_n$ são também uniformemente distribuídas. Este argumento é válido para qualquer outra combinação de n_1* shares *distintas. Sabemos que $s_2, s_3, \ldots, s_{n-1}$ são uniformemente distribuídas e independentes.*

Seja $s' = s_2 + s_3 + ... + s_{n-1}$ *para* $s_2, s_3, ..., s_{n-1}$ *fixos. Temos, de acordo com o algoritmo 2,*

$$\begin{aligned} s_n &= s - s_1 - (s_2 + ... + s_{n-1}) \\ &= (s - s') - s_1 \end{aligned} \tag{2.2}$$

Como s_1 continua uniformemente distribuído, para $s_2, ..., s_{n-1}$ fixos, concluímos que s_n é uniformemente distribuído e independente de $s_2, ..., s_{n-1}$.

Provado que u_3, v_3 são também uniformemente distribuídos, podemos sem perda de generalidade, simular a vista de uma única *party* para provar a segurança do protocolo. De notar que, todos os protocolos que consideramos têm como ponto inicial a operação de *share* (alg. 1) por parte da *input party* e , como ponto final, *desharing* (alg. 10) por parte da *output party*.

No mundo real (fig. 2.3), após a execução do protocolo π, a vista de $\mathcal{P}_1$ é a seguinte:

$$Vista_{\mathcal{P}_1} = \{u_1, v_1, r_{12}, r_{31}, u'_1, r'_{12}, r'_{31}, v'_1, u'_3, v'_3, w'_1, r''_{12}, r''_{31}, w_1\}$$

Dado que a *party* $\mathcal{P}_1$ foi corrompida, o adversário $\mathcal{A}$ tem acesso a esta vista e, também, aos *inputs* e *outputs* do protocolo, através do ambiente $\mathcal{Z}$. Sendo assim, a vista do adversário $\mathcal{A}$ é a concatenação da vista de $\mathcal{P}_1$ com $\{u, v, w\}$:

$$Vista_{\mathcal{A}} = \{u, v, u_1, v_1, r_{12}, r_{31}, u'_1, r'_{12}, r'_{31}, v'_1, u'_3, v'_3, w'_1, r''_{12}, r''_{31}, w_1, w\}.$$

É, agora, necessário descrever um simulador $\mathcal{S}$ que consiga reconstruir a vista de $\mathcal{P}_1$ no mundo ideal, de forma a que esta seja indistinguível para quaisquer *inputs* e *outputs*.

Simulador Sempre que $\mathcal{Z}$ requisita uma corrupção de uma *computing party* $\mathcal{P}_i$, $\mathcal{S}$ recebe os *inputs* de $\mathcal{P}_i$ especificados até ao momento, por $\mathcal{Z}$, e todos os *inputs* recebidos de $\mathcal{F}$. Agora, $\mathcal{S}$ irá usar esta informação para simular a vista completa da respectiva $\mathcal{P}_i$, participando no protocolo π e apresentado esta vista a $\mathcal{Z}$. Esta vista deve ser consistente com a que $\mathcal{A}$ tem nesse momento. Por fim, adicionamos $\mathcal{P}_i$ ao subconjunto de adversários $\mathcal{A}$.

Descrevemos agora os passos que o simulador $\mathcal{S}$ tem de executar para simular a vista de $\mathcal{P}_1$.

1. $\mathcal{S}$ pede à funcionalidade $\mathcal{F}$ o comprimento de n para determinar $\mathbb{Z}_{2^n}$
2. $\mathcal{S}$ gera u_1 e v_1 uniformemente distribuídos;
3. $\mathcal{S}$ gera r_{12} e r_{31} uniformemente distribuídos;
4. $\mathcal{S}$ calcula $u'_1 = u_1 + r_{12} - r_{31}$;
5. $\mathcal{S}$ gera r'_{12} e r'_{31} uniformemente distribuídos;
6. $\mathcal{S}$ calcula $v'_1 = v_1 + r'_{12} - r'_{31}$;
7. $\mathcal{S}$ gera u'_3 e v'_3 uniformemente distribuídos;
8. $\mathcal{S}$ calcula $w'_1 = u'_1 v'_1 + u'_1 v'_3 + u'_3 v'_1$;
9. $\mathcal{S}$ gera r''_{12} e r''_{31} uniformemente distribuídos;
10. $\mathcal{S}$ calcula $w_1 = w'_1 + r''_{12} - r''_{31}$;

Os passos {2,3},{4,5} e {8,9} correspondem à execução do algoritmo de *resharing* (alg 11) para u_1, v_1 e w'_1, respectivamente.

No mundo ideal (fig. 2.4), o simulador $\mathcal{S}$, pode simular a vista de $\mathcal{P}_1$, começando por gerar as *shares* $\{u_1, v_1\}$. Dado que, no mundo real, estes dois valores são gerados aleatoriamente em $\mathbb{Z}_{2^n}$, o simulador pode também gerá-los de forma aleatória. Para o fazer, este irá precisar de uma informação adicional, o comprimento de n, de forma a conseguir gerar estes valores em concordância com o protocolo π. O simulador requisita essa informação a $\mathcal{F}$ e gera as respectivas *shares* aleatoriamente.

Os valores $\{r_{12}, r_{31}, r'_{12}, r'_{31}, r''_{12}, r''_{31}\}$ que, no protocolo π, correspondem a valores gerados aleatoriamente durante o processo de *resharing*, podem também ser gerados pelo simulador de forma aleatória. Da vista de $\mathcal{P}_1$, restam as *shares* $\{u'_1, v'_1, w'_1, w_1\}$. No protocolo π, estas *shares* são construídas a partir dos valores, uniformemente distribuídos, já discutidos, da seguinte forma:

$$u'_1 = u_1 + r_{12} - r_{31}$$
$$v'_1 = v_1 + r'_{12} - r'_{31}$$
$$w'_1 = u'_1 v'_1 + u'_1 v'_3 + u'_3 v'_1$$

$$w_1 = w'_1 + r''_{12} - r''_{31}$$

O simulador $\mathcal{S}$ pode calcular estas *shares* da mesma forma, dado que ele já gerou, aleatoriamente, os valores necessários anteriormente. Como as *shares* são construídas a partir de valores uniformemente distribuídos, podemos afirmar que, também elas, são uniformemente distribuídas.

Concluímos assim, que para qualquer adversário $\mathcal{A}$, existe um simulador $\mathcal{S}$, tal que, a vista do mundo ideal é indistinguível da vista do mundo real. Como tal, o protocolo π realiza, com segurança, a funcionalidade ideal $\mathcal{F}$. As restantes provas de segurança, apresentadas no capítulo 3 desta dissertação, seguem este modelo de prova.

2.3. Sistemas de armazenamento de dados

São apresentados, neste capítulo, os dois principais paradigmas de sistemas de armazenamento de dados, o relacional e o não-relacional. O primeiro adotado há décadas, continua a ser amplamente utilizado, enquanto que o segundo, mais recente, tem-se tornado mais popular com o crescimento do *Big Data*.[3]

2.3.1. Sistemas de dados relacionais

Numa base de dados relacional, para além dos dados em si, são também armazenadas as suas relações. A noção de relações entre os dados armazenados permite melhorar a eficiência no acesso e pesquisa da base de dados. Estes sistemas de dados suportam, por norma, uma linguagem de *query* estruturada (*SQL*), que permite interagir com o sistema de dados, definindo e manipulando os dados. Outra característica importante dos sistemas de dados relacionais baseia-se nas garantias de atomicidade, consistência, isolamento e durabilidade (*ACID*) das suas transações.

Apresentamos de seguida, o sistema de dados relacional utilizado para o desenvolvimento do sistema de computação e armazenamento seguro.

[3]Termo geral que define conjuntos de dados tão grandes e complexos que se tornam difíceis de processar no paradigma de dados relacional.

Apache Derby

O *Apache Derby*[4] é um sistema de gestão de base de dados relacionais *open-source*, implementado totalmente em *Java* e disponível sob a licença *Apache*[5], versão 2.0. É muito leve, de fácil instalação e configuração, sendo facilmente integrado em projetos *Java*.

Por estas razões, escolhemos o *Apache Derby* para desenvolver o protótipo do sistema de armazenamento e computação segura, sobre o modelo relacional.

2.3.2. Sistemas de dados não-relacionais

Os sistemas de dados não-relacionais, vulgarmente conhecidos como *NoSQL*, surgiram da necessidade de processar volumes de dados com um crescimento exponencial. Como o próprio nome sugere, este tipo de sistemas de dados, não implementa o esquema típico, relacional, de tabela/chave e relações, a que o modelo relacional nos habituou. Em vez disso, estes sistemas providenciam esquemas próprios para a manipulação dos dados, de forma a satisfazer a corrente necessidade de serviços escaláveis.

Diferentes dos sistemas relacionais, que se regem pela propriedade *ACID*, os sistemas de dados não-relacionais, usam a propriedade *BASE* (*Basically Available, Soft-state, Eventually consistent*). Esta propriedade relaxa um pouco os requisitos de consistência, favorecendo assim a disponibilidade e o particionamento para obter uma melhor escalabilidade. Os sistemas de dados não-relacionais foram desenhados para responder a determinadas necessidades imediatas na manipulação de enormes quantidades de dados, mas ainda não são tão amplamente utilizados como os sistemas relacionais, apesar de terem sofrido melhorias nos últimos anos, ganhando cada vez mais importância.

Apresentamos de seguida o *HBase*, uma base de dados distribuída, não-relacional, que faz parte do projeto *open-source Apache Hadoop* e, sobre o qual, desenvolvemos o sistema de dados seguro apresentado no capítulo 4.

[4] http://db.apache.org/derby/
[5] http://www.apache.org/licenses/LICENSE-2.0.html

Apache HBase

O *HBase*[6] [10] é uma base de dados *NoSQL* orientada à coluna, escrita em *Java*, e que corre em cima de um sistema de ficheiros distribuído do *Hadoop*[7]. Ao contrário de sistemas de dados relacionais, o *HBase* não suporta, de forma nativa, uma linguagem de *query* estruturada (e.g., *SQL*). O *HBase* é facilmente integrável com a *framework MapReduce*, também do ecossistema *Hadoop*, para o processamento distribuído de dados.

Nas versões anteriores à 0.92, não era possível adicionar novas funcionalidades ao *HBase* sem ter de estender as classes base do mesmo, o que se podia tornar uma tarefa complicada. Nesta versão foram introduzidos os coprocessadores, uma *framework* que permite criar extensões flexíveis e genéricas dentro do *HBase*. Permite também computação distribuída sobre os dados armazenados com a ajuda do *MapReduce*.

Existem dois tipos de coprocessadores: os *observers* e os *endpoints*. Os *observers* assemelham-se aos *triggers* das bases de dados relacionais, permitindo que uma ação predefinida possa ser despoletada antes, ou depois, de uma determinada ação base do *HBase*. Por sua vez, os *endpoints* funcionam como as *stored procedures* das bases de dados relacionais, permitindo que uma ação predefinida possa ser invocada remotamente pelo cliente, quando este o entender. Estes dois tipos de coprocessadores podem ser livremente combinados para estender o *HBase* com funcionalidades arbitrariamente complexas.

A escolha deste sistema de dados não-relacional recai nestas propriedades apresentadas, nomeadamente, na fácil extensão das suas funcionalidades.

2.3.3. Sistemas de armazenamento de dados seguro

Apresentamos nesta secção o *CryptDB*, um sistema de armazenamento de dados seguro que consegue executar *queries* sobre dados cifrados. Deste sistema reutilizámos uma ideia, nomeadamente, a implementação de uma camada de *middleware* para converter as *queries* em operações passíveis de serem aplicadas sobre dados cifrados.

[6] http://www.hbase.apache.org/
[7] http://hadoop.apache.org/

CryptDB

O *CryptDB* [20, 19] é um projeto, desenvolvido pelo *MIT's Computer Science and Artificial Intelligence Lab*[8], que permite executar *queries SQL* sobre dados cifrados em aplicações suportadas por bases de dados relacionais. As *queries* são intercetadas por um *middleware* que tem como objectivo analisá-las e convertê-las nas operações correspondentes sobre dados cifrados. Não se trata, especificamente, de um sistema de computação seguro, como o *Sharemind*, mas sim, de um sistema de armazenamento seguro. Este projeto baseia-se em três ideias-chave: uma estratégia de encriptação *SQL-Aware* [21], uma encriptação ajustável à *query* e a utilização de camadas de encriptação.

As *queries SQL* são compostas por operadores primitivos, sendo os mais importantes as comparações de ordem e de igualdade, e as adições. Para a maioria destes operadores o *CryptDB* encontrou uma primitiva criptográfica que permite aplicá-los sobre os dados cifrados, sem conhecer a chave secreta. É esta a noção de estratégia de encriptação *SQL-Aware* em que, dadas estas primitivas criptográficas, cada entrada da base de dados é cifrada de forma a que seja possível executar os operadores *SQL* necessários.

A segunda ideia-chave é uma encriptação ajustável à *query*, isto é , o *CryptDB* ajusta o nível de encriptação de cada entrada da base de dados, durante a execução da *query*, de forma a preservar a privacidade dos dados. Em particular, o *CryptDB*, cifra, inicialmente, todos os dados com a primitiva criptográfica mais forte e, à medida que a aplicação executa *queries SQL*, o *CryptDB* ajusta a encriptação dos dados de forma a que possam ser realizadas as computações necessárias para satisfazer a *query*. Este modelo garante que os dados se encontrem sempre cifrados, com a máxima privacidade possível, mediante o tipo de *query* que é executado, e que não seja necessário declarar previamente qual o nível de encriptação necessário.

Por fim, a última ideia-chave passa por implementar a encriptação ajustável à *query* usando camadas de encriptação (*onion encryption*). Cada entrada da tabela é cifrada em várias camadas, começando por uma primitiva mais fraca, que permite certas computações, e indo até primitivas mais fortes, que não revelam qualquer tipo de informação sobre os dados, nem permitem qualquer tipo de computação como podemos ver na Figura 2.6.

[8] https://www.csail.mit.edu/

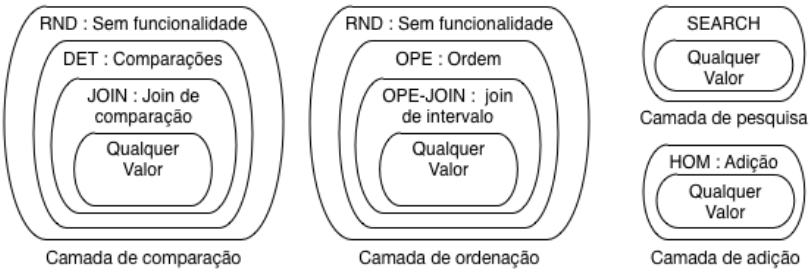


Figura 2.6.: Encriptação em camadas do *CryptDB*

Esta abordagem permite que os níveis de encriptação possam ser adequadamente ajustados do lado do servidor, evitando que o cliente precise de voltar a cifrar os dados. No entanto, este mecanismo de adequação do nível de encriptação faz com que, para executar determinadas *queries*, seja revelada alguma informação sobre os dados. O que pode ser revelado são as relações entre as entradas de uma tabela, como a ordem ou igualdade, nunca sendo revelado o conteúdo dos dados diretamente. A estrutura geral das tabelas também não é ocultada, sendo possível descobrir o número de entradas, o tipo das colunas ou até o tamanho aproximado dos dados em bytes.

O *CryptDB* apresenta-se como uma solução interessante para permitir o armazenamento de dados em servidores que não precisam de ser de confiança. Oculta, usando diversas primitivas criptográficas, a informação armazenada nas tabelas da base de dados, mantendo a possibilidade de se executarem *queries* sobre ela. Em termos de performance, segundo o autor, este sistema incorre numa redução de cerca de 27% em *throughput* [20] quando comparado com um sistema de dados sem encriptação. Posto isto, o *CryptDB* apresenta-se como uma solução prática para o armazenamento de dados sensíveis e/ou confidenciais.

Esta abordagem é diferente da abordagem de *Secure Multi-Party Computation* pois, no primeiro caso, toda a informação é armazenada e processada por uma só entidade enquanto que, no segundo, tudo é distribuído por várias entidades. No caso da *Secure Multi-Party Computation*, sabemos que uma só entidade nunca será capaz de recuperar os dados armazenados, se não existir conluio com outras partes, mesmo que detenha um poder computacional ilimitado. Já no caso do *CryptDB*, nunca saberemos se a entidade que armazena os dados terá poder computacional suficiente para decifrar os dados que tem na sua posse.

Capítulo 3

Algoritmos com segurança ajustável

De modo a suportar as *queries* descritas na secção 4.2, são necessários dois protocolos distintos para determinar relações de igualdade e desigualdade entre valores *secret shared*. Esses protocolos, *Equality(u,v)* (alg 4) e *GreaterThan(u,v)* (alg 7), podem ser encontrados no sistema de computação *Sharemind*. Para além desses dois protocolos, e como uma das contribuições, apresentamos duas versões alternativas de cada, que permitem, mediante um relaxamento das garantias de segurança e privacidade, melhorar o desempenho dos mesmos. Por outras palavras, estas versões alternativas permitem ajustar a segurança/velocidade dos algoritmos (um sempre em detrimento do outro). Para terminar, é realizada uma avaliação de segurança e é analisado o desempenho dos algoritmos propostos de modo a validar a sua utilidade num contexto real.

3.1. Determinar a relação de igualdade entre dois valores

Este primeiro algoritmo é utilizado para avaliar o predicado de igualdade entre dois valores partilhados através de *secret sharing*. Apresentamos de seguida a versão original do algoritmo, apresentada no *Sharemind* [4], e duas versões que oferecem um *tradeoff* entre a segurança e o desempenho do algoritmo.

3.1.1. Equal(u,v) - Versão *Sharemind*

Cada *computing party* tem, como *input*, uma *share* de um valor u e outra de um valor v e devolve uma *share* de um valor w, correspondente a 1 ou 0, para o caso de u e v serem iguais ou diferentes, respetivamente. A avaliação do predicado é realizada, de forma distribuída, entre as três *computing parties*. O algoritmo 4 começa por partilhar a diferença $u - v$ entre $\mathcal{P}_2$ e $\mathcal{P}_3$, como $e_2 + e_3$. De seguida resta determinar se $e_2 + e_3 = 0$, o que pode ser feito comparando e_2 e $-e_3$, de forma *bitwise*. Para tal, avaliamos a soma *bitwise* (XOR) de $\overline{p_1} = 2^n - 1 = 111...1$, $\overline{p_2}$ e $\overline{p_3}$. Podemos ver que $u = v$ se, e só se, todos os *bits* representados por $\overline{[\![p]\!]}$ são 1, o que é o caso quando a conjunção $[\![w]\!] = \wedge_{i=0}^{n-1} \overline{[\![p]\!]}^{(i)}$ é 1. É isto que é verificado com a invocação do algoritmo *BitConj* (Alg. 18).

Algoritmo 4: Protocolo $[\![w]\!] \leftarrow$ Equal($[\![u]\!]$, $[\![v]\!]$) para avaliar o predicado de igualdade.

Dados: Valores partilhados $[\![u]\!]$ e $[\![v]\!]$

Resultado: Valor $[\![w]\!]$ partilhado tal que w=1 se u=v, e 0 caso contrário.

1 $\mathcal{P}_1$ gera $r_2 \leftarrow \mathbb{Z}_{2^n}$ aleatório e calcula $r_3 \leftarrow (u_1 - v_1) - r_2$
2 $\mathcal{P}_1$ envia r_i a $\mathcal{P}_i$ (i=2,3)
3 $\mathcal{P}_1$ calcula $e_i = (u_i - v_i) + r_i$ (i=2,3)
4 $\mathcal{P}_1$ define $\bar{p}_1 \leftarrow 2^n - 1 = 111...1$
5 $\mathcal{P}_2$ define $\bar{p}_2 \leftarrow e_2$
6 $\mathcal{P}_3$ define $\bar{p}_3 \leftarrow (0 - e_3)$
7 Retorna $[\![w]\!] \leftarrow$ BitConj($\overline{[\![p]\!]}$)

Este algoritmo, quando aplicado sobre uma base de dados com milhares, ou até milhões, de entradas, pode resultar num enorme *overhead* de comunicação. Dado que o resultado obtido por cada *computing party* é uma *share* do *bit* que determina se u é igual a v, nenhuma *computing party* consegue saber realmente se u é igual a v, tendo por isso que devolver a sua *share* de u multiplicada pela sua *share* do resultado do algoritmo. Desta forma, aplicando isto a uma tabela de uma base de dados com n linhas, o cliente irá sempre receber, de cada *computing party*, todas as n linhas da tabela multiplicadas pelas *shares* do resultado do algoritmo 4. Como podemos ver, pode ser insuportável, ou não ser de todo prático, a nível de rede, devolver sempre as n linhas da tabela, quando os resultados pretendidos podem ser um subconjunto muito mais

pequeno de n. É para atenuar este problema de *overhead* que surge o algoritmo 5.

Prova de segurança

Seguindo o modelo de provas da secção 2.2, assumimos que existe um simulador $\mathcal{S}_{Equal}$, para qualquer adversário $\mathcal{A}$, que simula de forma indistinguível, no mundo ideal, a vista do mundo real. Como tal, assumimos que o protocolo (alg. 4) realiza, com segurança, a funcionalidade ideal $\mathcal{F}$, tal como foi mostrado em [7].

3.1.2. Equal(u,v) - Versão A

De forma a diminuir, consideravelmente, a quantidade de informação transmitida na rede, pelas três *computing parties* para o cliente, foram aplicadas algumas alterações ao algoritmo 4. Como vimos na secção anterior, o algoritmo 4 retorna, por cada *computing party*, uma *share* do valor w, que é 1 ou 0, que indica se u é, ou não, igual a v. Isto permite que nenhuma das *computing parties*, após executar o algoritmo, consiga aferir se o resultado foi 1 ou 0, preservando assim a privacidade dos dados.

A solução proposta passa por partilhar, entre as *computing parties*, a *share* do *bit* do resultado que obtiveram no passo 7, tornando assim esse valor público. Desta forma, cada *computing party* pode reconstruir o *bit* do resultado do algoritmo, sabendo efetivamente se o resultado foi 1 ou 0. Com esta informação adicional, cada *computing party* pode decidir se deve, ou não, devolver a linha em questão ao cliente, reduzindo assim consideravelmente o tráfego de dados na rede. No entanto, este passo revela alguma informação adicional às *computing parties* sobre os *inputs* u e v. Nomeadamente, revela quais as linhas da tabela que são iguais umas às outras, pois cada *computing party* só retornou um subconjunto do total das linhas da tabela, aquelas que são iguais a v. De notar que cada *computing party* continua sem saber o valor de u e de v, pois a publicação do resultado w da conjunção dos *bits* no passo 7, só revela se u é igual a v, nunca revelando os respetivos valores.

Algoritmo 5: Protocolo $[\![w]\!] \leftarrow$ EqualA($[\![u]\!]$, $[\![v]\!]$) para avaliar o predicado de igualdade.

Dados: Valores partilhados $[\![u]\!]$ e $[\![v]\!]$

Resultado: Valor $[\![w]\!]$ partilhado tal que w=1 se u=v, e 0 caso contrário.

1 $\mathcal{P}_1$ gera $r_2 \leftarrow \mathbb{Z}_{2^n}$ aleatório e calcula $r_3 \leftarrow (u_1 - v_1) - r_2$

2 $\mathcal{P}_1$ envia r_i a $\mathcal{P}_i$ (i=2,3)

3 $\mathcal{P}_i$ calcula $e_i = (u_i - v_i) + r_i$ (i=2,3)

4 $\mathcal{P}_1$ define $\bar{p}_1 \leftarrow 2^n - 1 = 111...1$

5 $\mathcal{P}_2$ define $\bar{p}_2 \leftarrow e_2$

6 $\mathcal{P}_3$ define $\bar{p}_3 \leftarrow (0 - e_3)$

7 $[\![w]\!] \leftarrow$ BitConj($\overline{[\![p]\!]}$)

8 $\mathcal{P}_1$ envia w_1 a $\mathcal{P}_i$ (i=2,3)

9 $\mathcal{P}_2$ envia w_2 a $\mathcal{P}_i$ (i=1,3)

10 $\mathcal{P}_3$ envia w_3 a $\mathcal{P}_i$ (i=1,2)

11 $w = (w_1 + w_2 + w_3) \mod 2$

12 Retorna w

Estas alterações, que podem ser vistas no algoritmo 5, permitem reduzir o tráfego gerado entre as *computing parties* e o cliente, com o custo de relaxar a privacidade sobre os dados do cliente. São apresentadas, mais à frente, as implicações destas alterações com maior detalhe.

Prova de segurança

Para mostrar que o protocolo π (alg. 5) realiza, com segurança, a funcionalidade ideal $\mathcal{F}$, comecemos por definir a funcionalidade:

- $\mathcal{F}$ recebe os *inputs* u e v da *input party*;
- $\mathcal{F}$ determina $w = 1$ caso $u = v$, $w = 0$ caso contrário.
- $\mathcal{F}$ retorna w à *output party*;

Para determinar a segurança deste algoritmo, que é em tudo igual ao algoritmo 4, mas com o passo adicional de tornar públicas as *shares* de w, de modo a reconstruir w, vamos começar por

determinar a vista da *party* $\mathcal{P}_1$, com base na vista (V_{Equal}) de $\mathcal{P}_1$ do algoritmo 4, que assumimos que é simulável por $\mathcal{S}_{Equal}$:

$$V_{\mathcal{P}_1} = V_{Equal} + \{w_2, w_3, w\}$$

Como podemos ver, para reconstruir o valor w, $\mathcal{P}_1$ precisa de receber, de $\mathcal{P}_2$ e $\mathcal{P}_3$, as *shares* w_2, w_3 que lhe faltam. Mostramos agora que, existe um simulador $\mathcal{S}$ capaz de simular a vista de $\mathcal{P}_1$.

Dado o simulador $\mathcal{S}_{Equal}$, que simula a vista V_{Equal}, construímos um novo simulador para simular a vista de $V_{\mathcal{P}_1}$.

Simulador:

1. Execução de $\mathcal{S}_{Equal}$
2. $\mathcal{S}$ pede w à funcionalidade ideal para pode gerar w_2, w_3 de forma coerente com o *output*
3. $\mathcal{S}$ gera w_2, w_3 tal que $w = w_1 + w_2 + w_3$

Como podemos ver, para que $\mathcal{S}$ consiga simular a vista de $\mathcal{P}_1$, de forma indistinguível, $\mathcal{S}$ começa por ter de gerar as *shares* w_2, w_3. $\mathcal{S}$ não pode fazer isso de forma aleatória pois pode não conseguir gerar valores que reconstruam w em concordância com o *output*. Como tal, o simulador $\mathcal{S}$ necessita de pedir à funcionalidade ideal $\mathcal{F}$, o valor de w. Esta informação adicional, w, representa se $u = v$ e, é libertada pelo protocolo.

Aplicado ao nosso sistema, isto representa que o algoritmo 5 revela quais as linhas que são iguais umas às outras, sem revelar o valor das mesmas.

Concluímos assim que, para qualquer adversário $\mathcal{A}$, existe um simulador $\mathcal{S}$ tal que, a vista do mundo ideal é indistinguível da vista do mundo real. Como tal, o protocolo π realiza, com segurança, a funcionalidade ideal $\mathcal{F}$.

3.1.3. Equal(u,v) - Versão B

Nos dois algoritmos até agora apresentados, a avaliação do predicado de igualdade é determinada pela conjunção dos *bits* das *shares*. Esta conjunção é realizada de forma privada, através

de multiplicações *bit* a *bit*. A operação de multiplicação, não sendo uma operação local, requer comunicação entre as *computing parties*, tornando assim o algoritmo computacionalmente mais dispendioso e, consequentemente, mais lento.

O que se procura nos armazenamento é uma rápida execução das operações, pelo que aplicámos algumas alterações ao algoritmo de forma a melhor significativamente o seu desempenho. Uma vez mais, esta melhoria na performance é feita em detrimento do nível de privacidade dos dados armazenados.

Algoritmo 6: Protocolo $[\![w]\!] \leftarrow$ EqualB($[\![u]\!]$, $[\![v]\!]$) para avaliar o predicado de igualdade.

Dados: Valores partilhados $[\![u]\!]$ e $[\![v]\!]$

Resultado: Valor $[\![w]\!]$ partilhado tal que w=1 se u=v, e 0 caso contrário.

1 $\mathcal{P}_1$ gera $r_2 \leftarrow \mathbb{Z}_{2^n}$ aleatório e calcula $r_3 \leftarrow (u_1 - v_1) - r_2$

2 $\mathcal{P}_1$ envia r_i a $\mathcal{P}_i$ (i=2,3)

3 $\mathcal{P}_i$ calcula $e_i = (u_i - v_i) + r_i$ (i=2,3)

4 $\mathcal{P}_2$ envia e_2 a $\mathcal{P}_1$, $\mathcal{P}_3$

5 $\mathcal{P}_3$ envia e_3 a $\mathcal{P}_1$, $\mathcal{P}_2$

6 $\mathcal{P}_i$ define w=1 se $e_2 == -e_3$, w=0 caso contrário (i=2,3)

7 Retorna w

No algoritmo 6 alterámos a forma como é avaliada a igualdade $e_2 = e_3$ entre $\mathcal{P}_2$ e $\mathcal{P}_3$. A comparação deixou de ser *bitwise*, permitindo que se consigam evitar todas as multiplicações da conjunção dos *bits* que, como já vimos, não são operações locais, sendo por isso dispendiosas. Neste algoritmo, $\mathcal{P}_2$ envia e_2 para $\mathcal{P}_3$ e este avalia se e_2 é, ou não, igual a $-e_3$ sendo, de seguida, enviado o resultado a $\mathcal{P}_1$ e $\mathcal{P}_2$.

Como a comparação já não é feita de forma *bitwise*, as *computing parties* conseguem determinar, para além de quantas e quais linhas são iguais umas às outras, a diferença entre u e v. Continuam, no entanto, sem saber qual o valor de u ou de v. Se este relaxamento na privacidade dos dados é, ou não, relevante, depende exclusivamente da sensibilidade dos dados armazenados e do uso que se pretende dar ao sistema de dados. Mais à frente, são apresentadas com mais detalhe, as implicações destas alterações no algoritmo 6.

Prova de segurança

Para mostrar que o protocolo π (alg. 6) realiza, com segurança, a funcionalidade ideal $\mathcal{F}$, comecemos por definir a funcionalidade:

- $\mathcal{F}$ recebe os *inputs* u e v da *input party*;
- $\mathcal{F}$ determina $w = 1$ caso $u = v$, $w = 0$ caso contrário.
- $\mathcal{F}$ retorna w à *output party*;

Agora, precisamos de especificar qual é a vista da *party* $\mathcal{P}_1$, e também a vista de $\mathcal{P}_2$ (que é simétrica com a de $\mathcal{P}_3$) ,para construirmos simuladores capazes de as reconstruir:

$$V_{\mathcal{P}_1} = \{u_1, v_1, r_2, r_3, e_2, e_3, w\}$$
$$V_{\mathcal{P}_2} = \{u_2, v_2, r_2, e_2, e_3, w\}$$

Iremos agora construir um simulador $\mathcal{S}_{\mathcal{P}_1}$ para simular a vista $V_{\mathcal{P}_1}$:

Simulador

- $\mathcal{S}_{\mathcal{P}_1}$ gera u_1, v_1 uniformemente distribuídos
- $\mathcal{S}_{\mathcal{P}_1}$ gera r_2 uniformemente distribuído
- $\mathcal{S}_{\mathcal{P}_1}$ calcula $r_3 = (u_1 - v_1) - r_2$
- $\mathcal{S}_{\mathcal{P}_1}$ pede a $\mathcal{F}$ o valor de w
- $\mathcal{S}_{\mathcal{P}_1}$ gera aleatoriamente $e_2 = e_3$ se $w = 1$, $e_2 != e_3$ se $w = 0$

Como podemos ver, para que $\mathcal{S}_{\mathcal{P}_1}$ consiga simular, de forma indistinguível , a vista de $\mathcal{P}_1$, este precisa de recorrer à funcionalidade ideal $\mathcal{F}$ para obter o valor de w e assim conseguir gerar os valores e_2 e e_3 de forma consistente com o *output*.

A informação adicional que este algoritmo liberta é a diferença entre u e v, sem revelar u nem v individualmente. Isto acontece porque o adversário tem na sua vista os valores e_2 e e_3, que permitem determinar a diferença entre u e v.

Para a vista de $\mathcal{P}_2$, assumimos a existência de um simulador semelhante, capaz de reproduzir a vista do mundo real de forma indistinguível no mundo ideal. Este simulador difere do anterior, num único ponto, que é na não necessidade de gerar r_3.

Concluímos assim que, para qualquer adversário $\mathcal{A}$, existe um simulador $\mathcal{S}$ tal que, a vista gerada no mundo ideal é indistinguível da vista do mundo real. Como tal, o protocolo π realiza, com segurança, a funcionalidade $\mathcal{F}$.

3.2. Determinar a relação de desigualdade entre dois valores

Este protocolo serve para avaliar o predicado de desigualdade entre dois valores *secret shared*, u e v. O protocolo *GreaterThan* é baseado na observação de que podemos determinar a relação entre dois valores, extraindo o *bit* mais significativo da sua diferença. Esta abordagem é restrita a casos onde o *bit* mais significativo é interpretado como *bit* de sinal. Desta forma, quando aplicado sobre inteiros de 32 *bits* sem sinal, acontece uma comparação de 31 *bits* sem sinal ou uma comparação de 32 *bits* com sinal.

3.2.1. GreaterThan(u,v) - Versão *Sharemind*

Neste protocolo, o bit mais significativo é extraído usando o algoritmo 17. Este algoritmo aplica um *shift* para a direita, ao *bit* mais significativo da diferença (u-v), de forma a torná-lo no bit menos significativo. Este bit pode ser invertido, de forma a transformar este protocolo, que avalia o predicado de maior, num que avalia o predicado de menor-ou-igual.

Calcular um *shift* de p posições para a esquerda seria trivial, bastando multiplicar a *share* local pela constante pública 2^p. Tratando-se de um *shift* para a direita, torna-se mais complicado por não sabermos qual é *carry* do *overflow* módulo 2^n. É, por isso, necessário uma rotina que determine o *overflow* (Alg. 15). Para calcular o *overflow* mais facilmente, é feito um *resharing*, temporário dos inputs, de forma a que estes fiquem distribuídos por só duas *computing parties*.

Para tal é utilizado o protocolo *ReshareToTwo* (Alg. 12). Este *resharing*, entre duas *computing parties*, torna o processo mais fácil pois garante que o *overflow* será 0 ou 1.

Algoritmo 7: Protocolo $[\![w]\!] \leftarrow$ GreaterThan($[\![u]\!]$, $[\![v]\!]$) para avaliar o predicado de maior ou igual.

Dados: Valores partilhados $[\![u]\!]$ e $[\![v]\!]$

Resultado: Valor $[\![w]\!]$ partilhado tal que w=1 se u>v, e 0 caso contrário.

1 $\mathcal{P}_i$ calcula $d_i \leftarrow u_i - v_i$

2 $[\![w']\!] \leftarrow ShiftRight([\![\overline{d}]\!], 31)$

3 Retorna $[\![w]\!] \leftarrow Reshare([\![w']\!])$

Com as rotinas para redistribuir as *shares* e para calcular o *overflow*, é possível executar o algoritmo *ShiftRight* e, consequentemente, o algoritmo 7 que determina o predicado de desigualdade. Este protocolo é muito mais lento do que o algoritmo 4 porque, para calcular o *overflow*, são necessárias muitas multiplicações que, como já vimos, não são operações locais e, por isso, requerem rondas de comunicação entre as *computing parties*. No sentido de tentar melhorar o desempenho deste algoritmo, surgem os algoritmos 8 e 9.

Prova de segurança

Seguindo o modelo de provas da secção 2.2, assumimos que existe um simulador $\mathcal{S}$, para qualquer adversário $\mathcal{A}$, que simula de forma indistinguível, no mundo ideal, a vista do mundo real. Como tal, assumimos que o protocolo (alg. 7) realiza, com segurança, a funcionalidade ideal $\mathcal{F}$.

3.2.2. GreaterThan(u,v) - Versão A

Tal como no algoritmo 4, foram aplicadas algumas alterações ao algoritmo 7, de forma a reduzir consideravelmente o tráfego de dados na rede, entre o cliente e as *computing parties*. À semelhança do que acontece no algoritmo 4, o *output* do algoritmo 7, em cada *computing party*, é a *share* do bit que representa, se $u > v$. Desta forma, e como já foi referido, as *computing*

parties não sabem efectivamente se $u > v$, sendo obrigadas a retornar todas as linhas da sua tabela, multiplicadas pela *share* do resultado, cabendo ao cliente a tarefa de abrir as *shares* e verificar quais as linhas em que $u > v$.

Para contornar este problema, aplica-se o mesmo conceito aplicado no algoritmo 5, permitindo às *computing parties* abrir as *shares* dos resultados da comparação de desigualdade e, assim, saber quais as linhas que deve retornar ao cliente. Este passo, como já foi dito, revela alguma informação adicional sobre os inputs, mas permite reduzir consideravelmente o tráfego na rede, entre o cliente e as *computing parties*. Podemos ver no algoritmo 8 as alterações aplicadas.

A prova de segurança deste algoritmo é análoga à prova do algoritmo 5 e, por isso, não a apresentaremos aqui. Neste caso, a informação adicional que o algoritmo liberta é se u é maior do que v, sem revelar nunca o valor quer de u, quer de v.

Algoritmo 8: Protocolo $[\![w]\!] \leftarrow$ GreaterThanA($[\![u]\!]$, $[\![v]\!]$) para avaliar o predicado de maior ou igual.

Dados: Valores partilhados $[\![u]\!]$ e $[\![v]\!]$

Resultado: Valor $[\![w]\!]$ partilhado tal que w=1 se u>v, e 0 caso contrário.

1 $\mathcal{P}_i$ calcula $d_i \leftarrow u_i - v_i$
2 $[\![w']\!] \leftarrow ShiftRight([\![\overline{d}]\!], 31)$
3 $[\![w]\!] \leftarrow Reshare([\![w']\!])$
4 $\mathcal{P}_1$ envia w_1 a $\mathcal{P}_i$ (i=2,3)
5 $\mathcal{P}_2$ envia w_2 a $\mathcal{P}_i$ (i=1,3)
6 $\mathcal{P}_3$ envia w_3 a $\mathcal{P}_i$ (i=1,2)
7 $w = (w_1 + w_2 + w_3) \mod 2^n$
8 Retorna w

3.2.3. GreaterThan(u,v) - Versão B

Tal como no algoritmo que determina o predicado de igualdade, o trabalho mais pesado deste algoritmo reside no elevado número de multiplicações que é necessário efetuar para realizar operações sobre os *bits* das *shares*, não revelando assim informação sobre os *inputs* às

computing parties. É possível aplicar algumas alterações a este algoritmo, de forma a que as computações não sejam realizadas de forma *bitwise*, reduzindo assim, consideravelmente, o número de multiplicações necessárias.

Propomos determinar a diferença $u - v$, sem recorrer ao cálculo do *bit* mais significativo, através do algoritmo 17. Desta forma, eliminamos todas as multiplicações necessárias anteriormente, sendo a diferença calculada localmente e depois partilhada entre as *computing parties*, de forma a que cada *party* saiba quais são as linhas da tabela que cumprem o requisito do predicado de desigualdade.

Estas alterações, como seria de esperar, melhoram significativamente o desempenho do algoritmo revelando alguma informação adicional acerca dos *inputs* armazenados. Mais concretamente, para além de revelar as linhas que são maiores ou menores que um determinado valor v desconhecido, revela agora também a diferença entre u e v. No entanto, o valor concreto de u e de v continuam secretos.

Não iremos apresentar a prova de segurança deste algoritmo, pois esta é semelhante à prova apresentada para o algoritmo 6.

Algoritmo 9: Protocolo $[\![w]\!] \leftarrow$ GreaterThanB($[\![u]\!]$, $[\![v]\!]$) para avaliar o predicado de maior ou igual.

Dados: Valores partilhados $[\![u]\!]$ e $[\![v]\!]$

Resultado: Valor $[\![w]\!]$ partilhado tal que w=1 se u>v, e 0 caso contrário.

1 $\mathcal{P}_i$ calcula $d_i \leftarrow u_i - v_i$

2 $\mathcal{P}_1$ envia d_1 a $\mathcal{P}_i$ (i=2,3)

3 $\mathcal{P}_2$ envia d_2 a $\mathcal{P}_i$ (i=1,3)

4 $\mathcal{P}_3$ envia d_3 a $\mathcal{P}_i$ (i=1,2)

5 $\mathcal{P}_i$ define w=1 se $d_1 + d_2 + d_3 > 0$, w=0 caso contrário

6 Retorna w

Capítulo 4

Validação experimental

Apresentamos, neste capítulo, os detalhes de implementação mais relevantes que levaram ao desenvolvimento do sistema proposto. Discutimos a arquitetura adotada, quer para o sistema relacional, quer para o não-relacional, e também o subconjunto de *queries* atualmente suportadas e implementadas.

Por fim, na secção 4.5, discutimos e analisamos o impacto da integração de algoritmos de *secret sharing*, em sistemas de armazenamento de dados, com base nos *tradeoffs* discutidos no capítulo 3.

4.1. Arquitetura do sistema

A implementação do sistema de dados seguro pode ser divida em duas partes. Numa primeira temos a implementação sobre bases de dados relacionais e, numa segunda, a implementação sobre bases de dados não-relacionais. Ambas as implementações têm implementados os mesmos algoritmos de *secret sharing* discutidos no capítulo 3. A arquitetura de ambos os sistemas é muito semelhante, variando apenas na maneira como os algoritmos foram introduzidos nos sistemas de dados, e na maneira como as *queries* são interpretadas. Discutimos, agora, a estrutura de cada implementação separadamente.

No caso da implementação em sistemas de dados relacionais, temos a seguinte estrutura:

um cliente que fornece os seus dados (*inputs*) para serem armazenados, e que executa *queries* (pede *outputs*) sobre eles; três bases de dados relacionais, que serão as *computing parties* do sistema, e que comunicam entre si, num modelo de *Secure Multi-Party Computation*, para executar as *queries* lançadas pelo cliente, de forma segura. Neste sistema a *input party* é também a *output party*. Estas bases de dados têm implementados os algoritmos de *secret sharing* apresentados no capítulo 3. Podemos observar, na figura 4.1, como estes componentes se relacionam, implementados com três base de dados *Apache Derby* e um cliente *Java*.

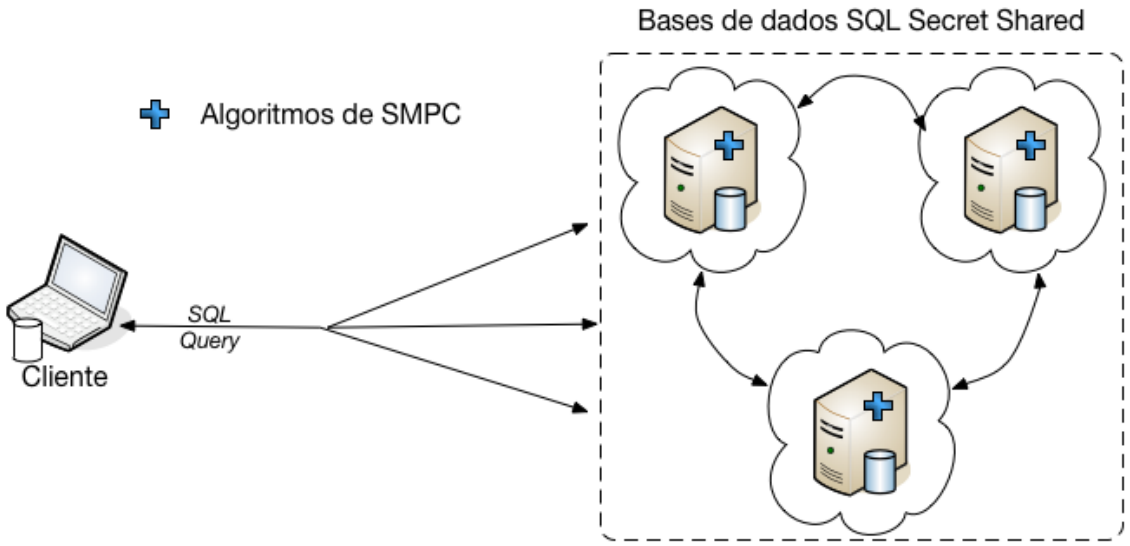


Figura 4.1.: Arquitetura do sistema em *Derby*

A implementação em sistemas de dados não-relacionais, com coprocessadores, é semelhante à apresentada para os sistemas de dados relacionais. Neste caso, dado que os sistemas de dados não-relacionais não suportam, por defeito, uma linguagem de *query* estruturada, adicionamos um *middleware* que adiciona esta possibilidade. Este *middleware*, designado por *SQL Engine* [25], irá funcionar como um conversor de instruções *SQL* para operações passíveis de serem interpretadas em sistemas não-relacionais. Segue, na figura 4.2, um esquema que mostra as relações entre os componente desta implementação.

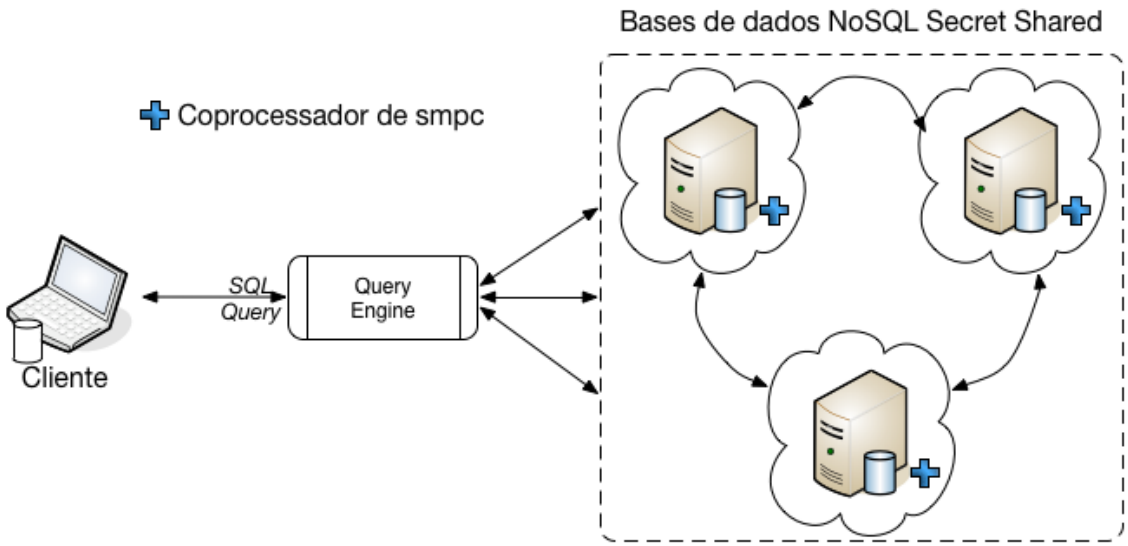


Figura 4.2.: Arquitetura do sistema em *HBase*

4.2. *Queries* suportadas

Para esta avaliação experimental, implementamos um subconjunto de *queries SQL* tipicamente utilizadas. Mais concretamente, são suportadas cláusulas do tipo ”SELECT WHER”,”INSERT INTO” e ”DELET”. Os tipos de dados suportados são inteiros e strings. As *strings* são, internamente, representadas como inteiros (convertendo para BigInt do *Java*) para permitir a execução de operações sobre elas. As operações suportadas para a cláusula ”WHERE” são as seguintes : $=, \neq, <, >, \leq e \geq$. Estas operações permitem determinar relações de igualdade e/ou desigualdade entre inteiros.

Este limitado subconjunto de operações parece-nos suficiente para demonstrar a aplicabilidade de protocolos de *Secure Multi-Party Computation*, em sistemas de dados com suporte para linguagens de *query* estruturadas.

Exemplos:

SELECT name FROM table WHERE $age = 23$;

SELECT age FROM table WHERE *name* =’Alice’;

SELECT * FROM table WHERE age$\geq$ 23;

INSERT (id,name,age) INTO table;

DELETE FROM table WHERE age$<$ 18;

É importante lembrar que, sempre que um utilizador lança uma *query*, os nomes das tabelas, nomes das colunas e tamanho das tabelas são públicos. Já os comparadores das cláusulas do tipo ”WHERE”, são *secret shared* e, por isso, privados. Pegando no primeiro exemplo, o campo privado seria o valor ’23’. Um campo é tornado privado aplicando-lhe um algoritmo de *secret sharing*. Exemplificando para o mesmo exemplo, o que cada uma das *computing parties* receberia, seria:

$\mathcal{P}_1$: SELECT name FROM table WHERE $age = 11$;

$\mathcal{P}_2$: SELECT name FROM table WHERE $age = 19$;

$\mathcal{P}_3$: SELECT name FROM table WHERE $age = 25$;

Assume-se, neste último exemplo, que o algoritmo de *secret sharing* opera sobre o anel $\mathbb{Z}_{32}$. Desta forma, temos:

$$(11+19+25) \mod 32 \equiv 23$$

É possível suportar outros tipos de dados, tais como vírgula flutuante [14, 16], armazenando, separadamente, a sua mantissa, base e expoente como inteiros. É também possível realizar outro tipo de *queries*, tais como *joins* [17], e suportar outro tipo de operações. Estes tipos de dados e *queries* não fazem parte desta avaliação experimental, pertencendo ao trabalho futuro.

4.3. Especificação do sistema

Para este sistema, pretende-se que sejam disponibilizadas, ao utilizador do sistema de dados, garantias de segurança e privacidade quanto aos dados armazenados. Para tal, vão ser utilizados os algoritmos propostos em [4] com algumas alterações, descritas no capítulo 3, para ajustar o sistema às necessidades do utilizador.

Introduzir garantias de privacidade sobre os dados armazenados, usando *secret sharing*, implica, como seria de esperar, um decréscimo da performance no acesso e nas operações sobre os dados. Esta degradação de performance, quando comparada com uma versão *vanilla* de um sistema de dados, é significativa. Em certos casos, isto pode não ser aceitável e, para atenuar isso, implementamos os algoritmos apresentados em 3, que beneficiam o desempenho, em detrimento da segurança e privacidade. Por outro lado, caso o tempo de execução não seja um problema, ou a total privacidade dos dados seja o fator prioritário, podem ser usadas as versões mais seguras dos algoritmos.

É também um requisito que a integração das técnicas de *Secure Multi-Party Computation*, no sistema de dados, seja totalmente transparente ao utilizador. Isto é, pretende-se que todo o processo de interação do utilizador com o sistema de dados seguro (introdução, consulta e remoção dos dados) seja transparente. O utilizador deve utilizar o sistema como que se de uma simples base de dados se tratasse e não como um sistema seguro e distribuído por três entidades distintas.

Adicionalmente, pretende-se que, quer no modelo relacional, quer no não-relacional, se consiga consultar o sistema de dados usando *queries SQL*, por estas serem amplamente utilizadas há décadas.

4.4. Implementação

Descrevemos, nesta secção, a abordagem adotada para a implementação das técnicas de *Secure Multi-Party Computation* nos sistemas de dados relacionais e nos não-relacionais.

4.4.1. Implementação em bases de dados relacionais

A implementação num sistema de dados relacional foi feita sobre o *Apache Derby*. Ao contrário do *HBase*, que suporta um mecanismo de coprocessadores, no *Derby*, foi necessário alterar algumas linhas de código diretamente nas suas classes, de forma a suportar as novas operações de *secret sharing*. Esta implementação direta no motor de base de dados é mais complexa do que a implementação via coprocessadores, pois exige um conhecimento mais profundo do sistema de dados.

Primeiramente, implementamos, em Java, os algoritmos de *secret sharing* apresentados no capítulo 3 e nos anexos, em cada uma das três *computing parties*, para que o *Derby* os possa utilizar. De seguida, efetuando algumas alterações nas classes que implementam o *table scan* e o *index scan*, *BulkTableScanResultSet* e *TableScanResultSet*, interceptamos as *queries* que chegam ao *Derby*, interpretamos e extraímos os parâmetros das mesmas (operadores, comparadores, campos, tabela).

Dependendo do operador extraído, instruímos o *Derby* a aplicar um determinado algoritmo de *secret sharing*, usando como parâmetro, o comparador passado na *query* e as entradas da tabela especificada. Por fim, como se de uma *query* normal se tratasse, é devolvido o resultado ao cliente. Para executar os algoritmos adicionados, é necessário, após interceptar a *query*, inibir o *Derby* de executar ele próprio a *query* localmente. Isto é, não queremos que ao lançar, por exemplo, a *query "SELECT * FROM table WHERE age>18"* , cada uma das bases de dados (*computing parties*), execute esta *query* sobre os seus dados, mas sim que, em conjunto, apliquem os algoritmos de *secret sharing* sobre os seus dados.

4.4.2. Implementação em bases de dados não-relacionais

O mecanismo de coprocessadores do *HBase*, que permite estender facilmente as suas funcionalidades, facilita a integração de técnicas de *Secure Multi-Party Computation* quando comparado com a integração no *Derby*.

Optamos por um coprocessador do tipo *observer*, mais especificamente um *BaseRegionObserver*, que se assemelha a um *trigger* numa base de dados relacional. Este tipo de coprocessador pode observar e mediar operações invocadas pelo cliente no *HBase* antes, ou

depois, de serem executadas numa determinada região, tais como o *get, put, delete, scan*.

São relevantes, para este coprocessador, as ações *scannerOpen* e *scannerNext*, que acontecem quando é invocado um *scan*. No *scannerOpen*, intercetamos os operadores e comparadores que foram lançados na *query*, e que vêm como um filtro no *HBase*, após passarem pela *SQL Engine*, para o *scanner*. No *scannerNext*, intercetamos as linhas da tabela para lhes aplicar os algoritmos de *secret sharing*, antes de estas serem devolvidas ao cliente. Ao intercetar o filtro, guardamos os valores e colocamos o filtro a nulo, para que o *HBase* não aplique o filtro convencional e nos deixe aplicar os algoritmos de *secret sharing*. De seguida, sempre que o *scanner* for buscar mais linhas à tabela, invocando o *scannerNext*, serão, novamente aplicados, os algoritmos de *secret sharing* com os operadores e comparadores, previamente extraídos do filtro. Por fim, o *HBase* retorna os resultados da *query* ao cliente.

4.5. Análise de desempenho

4.5.1. Considerações gerais

A aplicação de técnicas de *secret sharing* sobre os dados armazenados faz com que a visão das três *computing parties*, sobre os dados que cada uma armazena, não seja a mesma, dado que as *shares* foram geradas aleatoriamente. Por este motivo, com esta implementação, não é possível executar um *index scan* numa tabela, sendo necessário executar sempre um *table scan*.

Como a chave primária, no caso do *Derby*, ou a *row key*, no caso do *HBase*, tem de ser única, torna-se complicado gerar as suas *shares* para tabelas muitos grandes, evitando colisões. Dado que, para uma mesma linha numa tabela, vamos ter diferentes chaves-primárias em cada base de dados, seja ela relacional ou não-relacional, não é possível executar um *index scan*. Isto resulta numa degradação de *performance* porque, os *index scans* são mais rápidos que os *table scans*. A razão por trás disto, reside no facto de que, num *index scan*, só são percorridas as linhas que têm um índice dentro de um determinado intervalo. Já num *table scan* é sempre necessário percorrer toda a tabela.

Por último, referimos que, as *queries* foram processadas em *batch*, de 16[1] linhas de cada vez.

[1] Este valor vem por defeito no *Derby*

Desta forma, conseguimos minimizar as rondas de comunicação necessárias para executar os algoritmos de *secret sharing* e, assim, melhorar o desempenho geral do sistema.

ID	NOME	IDADE
1	187237	23124119
2	8972347	53627181
3	1238974	123134991
...	...	...
500000	53487923	3425172

Tabela 4.1.: Exemplo da estrutura da tabela da base de dados

Na tabela 4.1 podemos ver a estrutura da tabela utilizada, em cada *computing party*, para testar a execução de *queries* sobre o sistema de armazenamento e computação segura. Os valores nas colunas ”Nome” e ”Idade” encontram-se na forma *secret shared*.

4.5.2. Ambiente e configuração

Descrevemos nesta secção, as ferramentas, linguagens de programação e configurações usadas para implementar os algoritmos de *secret sharing* e, para desenvolver e testar os sistemas de computação e armazenamento seguro.

- **Especificações da máquina**
  - Processador : 1,8 GHz Intel Core i5
  - Memória : 8 GB 1600 MHz DDR3
  - Sistema operativo : OS X 10.9.4 (13E28)
- **Linguagens**
  - Javac 1.8.0_20
- **Sistemas de armazenamento de dados**

  - Apache Derby 10.10
  - Apache HBase 0.98.1-cdh5.1.2
- **Gerador de números pseudo-aleatórios**
  - *SecureRandom* do Java com a interface SHA1PRNG do *provider* SUN.

4.5.3. Tempos de execução

As métricas apresentadas nesta secção foram obtidas para a execução de uma determinada *query* sobre uma base de dados com uma tabela de 500000 entradas, com a estrutura apresentada na tabela 4.1. As medições incorporam todo o processo, desde o lançar da *query* pelo cliente, até à reconstrução das *shares* do resultado, também pelo cliente. São apresentados os tempos de execução de todas as versões dos algoritmos apresentados no capítulo 3 e de uma versão *vanilla*[2] do *derby*.

Igualdade

As *queries* usadas para avaliar o predicado de igualdade foram *queries* com a seguinte estrutura:

SELECT * FROM table WHERE age$= x$
SELECT * FROM table WHERE age$\neq x$

onde x é um inteiro arbitrário. Este tipo de *query* usa um dos seguintes algoritmos: 4,5 ou 6; cujos desempenhos queremos avaliar.

Algoritmo	Segundos	Minutos	Ratio
Equal(u,v) (alg. 4)	3310	55.17	1
EqualA(u,v) (alg. 5)	3140	52.33	1.05
EqualB(u,v) (alg. 6)	45	0.75	73.56
Sem *secret sharing*	0.42	0.01	-

Tabela 4.2.: Tempo de execução de uma *query* que avalia um predicado de igualdade em 500000 linhas

[2]Vanilla : Diz-se de uma versão de um *software* que não sofreu alterações relativamente à sua versão original

Na tabela 4.2 podemos ver o tempo de execução, sobre uma tabela de 500 mil linhas, de uma *query* que avalia um predicado de igualdade. Apresentamos o tempo de execução para as diferentes versões do algoritmo *Equality(u,v)* discutidas no capítulo 3 e, também, o tempo de execução, da mesma *query*, sobre uma só base de dados sem *secret sharing*.

Comecemos por analisar o tempo de execução da versão *vanilla*, sem *secret sharing*, do *Derby*. Para este caso temos um tempo de execução inferior a um segundo, mais concretamente 0.42 segundos. Este é o valor de referência, ao qual é desejável aproximar os tempos dos restantes algoritmos.

Para a versão do *Sharemind* do algoritmo *Equal(u,v)* (alg. 4), em que todas as linhas da tabela são devolvidas ao cliente (umas com os valores que satisfazem o predicado, as restantes a zero), registámos o pior tempo de execução. Este algoritmo demora, em média, 3310 segundos a executar a *query*. Este é um valor muito alto (quase uma hora), quando comparado com a versão *vanilla*, mas é o algoritmo que nos dá as maiores garantias de segurança e privacidade.

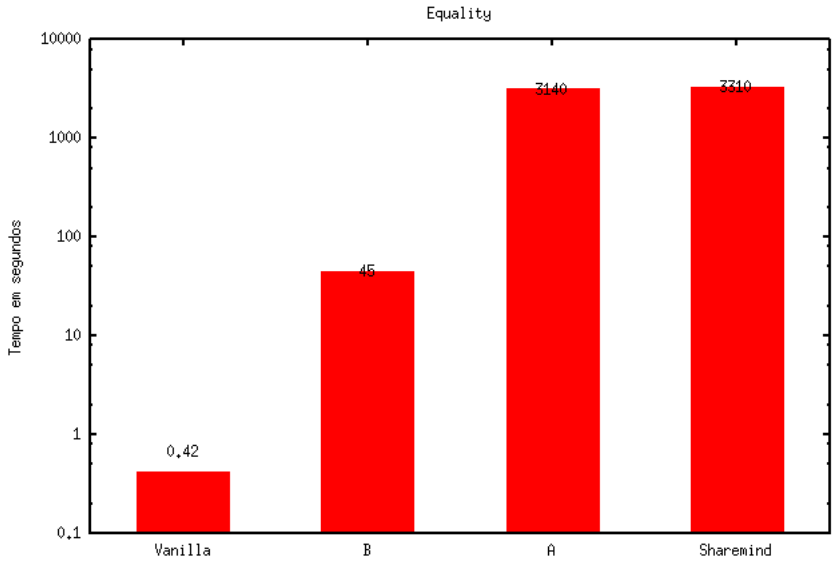


Figura 4.3.: Tempo de execução de uma *query* que avalia um predicado de igualdade em 500000 linhas

Para o algoritmo 5 (A), registamos um tempo de execução de 3140 segundos. Este valor, ligeiramente inferior ao anterior, deve-se à alteração aplicada ao algoritmo, que faz com que

só as linhas que satisfazem o predicado de igualdade sejam retornadas ao cliente. Com isto, reduziu-se a quantidade de dados transferida na rede (não contemplado nestas medições) e reduziu-se a quantidade de reconstruções de *shares* por parte do cliente. Esta redução de reconstruções de *shares* traduz-se na ligeira melhoria observada no tempo de execução.

Finalmente, para o algoritmo 6 (B), registamos um tempo de execução de 45 segundos, o que se traduz numa melhoria na ordem das 73 vezes, comparando com o algoritmo 4. Esta melhoria significativa deve-se às alterações aplicadas ao algoritmo, de forma a evitar as multiplicações bit-a-bit, que são operações de elevado custo computacional. Relembramos que, nesta versão do algoritmo, não temos garantias de segurança e privacidade tão estritas, permitindo que sejam reveladas algumas informações sobre os dados, como referimos no capítulo 3.

Na figura 4.3 podemos ver, graficamente, as relações entre o tempo de execução das diferentes versões do algoritmo. De notar que a escala do tempo de execução é logarítmica para facilitar a visualização dos valores.

Desigualdade

As *queries* usadas para avaliar o predicado de desigualdade foram *queries* com a seguinte estrutura:

SELECT * FROM table WHERE age$> x$
SELECT * FROM table WHERE age$\geq x$
SELECT * FROM table WHERE age$< x$
SELECT * FROM table WHERE age$\leq x$

onde x é um inteiro arbitrário. Este tipo de *query* usa um dos seguintes algoritmos: 7,8 ou 9; cujos desempenhos queremos avaliar.Os números apresentados, representam a média do tempo de execução destas quatro *queries*.

Algoritmo	Segundos	Minutos	Ratio
GreaterThan(u,v) (alg. 7)	36975	616.25	1
GreaterThanA(u,v) (alg. 8)	34710	578.5	1.06
GreaterThanB(u,v) (alg. 9)	41	0.68	901.83
Sem *secret sharing*	0.42	0.01	-

Tabela 4.3.: Tempo de execução de uma *query* que avalia um predicado de desigualdade em 500000 linhas

A tabela 4.3 exibe os tempos de execução, sobre a mesma tabela de 500000 linhas, de uma *query* que avalia um predicado de desigualdade. Apresentamos os tempos de execução para as diferentes versões do algoritmo *GreaterThan(u, v)*, discutidas no capítulo 3 e, novamente, o tempo de execução, da mesma *query*, sobre uma versão *vanilla* do *Derby*.

Quanto ao tempo de execução da versão *vanilla*, podemos ver que este é igual ao apresentado para o predicado de igualdade, na tabela 4.2. De notar que, sem *secret sharing*, a avaliação do predicado de igualdade e de desigualdade requer, sensivelmente, o mesmo tempo computacional.

Relativamente à versão do *Sharemind* do algoritmo GreaterThan (alg .4), obtemos, como no caso do algoritmo 4, o pior tempo de execução. Mais concretamente, medimos um tempo de execução de 36975 segundos (mais de 10 horas). Este valor é, como seria de esperar, muito superior ao medido para a versão *vanilla* mas é, também, cerca de 10 vezes superior ao apresentado pelo algoritmo 4. Isto deve-se ao facto de a complexidade deste algoritmo ser maior e exigir um significativo número superior de operações de multiplicação, para avaliar o predicado de desigualdade. Este é o algoritmo que oferece os níveis mais altos de segurança.

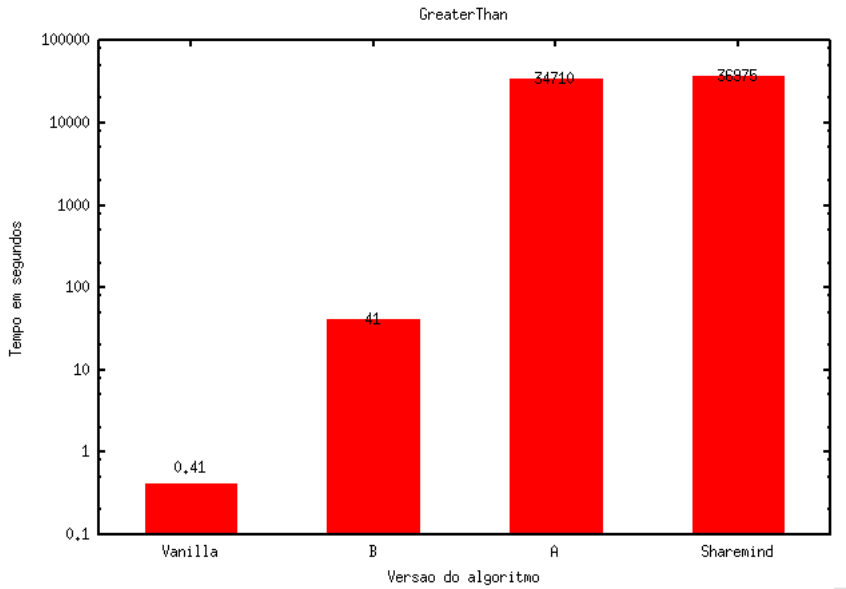


Figura 4.4.: Tempo de execução de uma *query* que avalia um predicado de desigualdade em 500000 linhas

Para o algoritmo 8 (A), foi registado um tempo de execução de 34710 segundos. Mais uma vez, esta ligeira melhoria, quando comparada com o algoritmo 7, deve-se à alteração no algoritmo para só retornar as linhas que satisfazem o predicado de desigualdade.

Por fim, o algoritmo 9 (B) apresenta o melhor tempo de execução dos algoritmos que avaliam o predicado de desigualdade. Para este algoritmo, medimos um tempo de 41 segundos, o que é cerca de 900 vezes menos do que no algoritmo 7. Ao deixar de determinar a diferença $u - v$, de forma *bitwise*, conseguimos um ganho muito grande em performance. Em contrapartida, tal como no algoritmo 6, este algoritmo revela informações adicionais sobre os dados, quando comparado com o 7, como já descrevemos no capítulo 3.

Como foi possível observar, a introdução de algoritmos de *secret sharing* nos sistemas de dados e as inerentes computações dispendiosas, fazem com que os tempos de execução aumentem significativamente. Em alguns casos, estes tempos são aceitáveis, mas na maioria, não o são. As versões mais rápidas que apresentamos, para ambos os algoritmos que avaliam os predicados de igualdade e desigualdade, são alternativas a ter em conta, pois têm uma

performance que, ainda que distante, se consegue aproximar mais das performances dos sistemas de dados que não implementam técnicas de *secret sharing* e *Secure Multi-Party Computation*.

Capítulo 5

Conclusão

Desenvolvemos, recorrendo a técnicas e conceitos de *secret sharing*, um sistema de armazenamento e computação segura que funciona, quer em bases de dados relacionais, quer em não-relacionais, suportando, a linguagem de *query SQL*. No caso do sistema relacional, as alterações foram implementadas diretamente no motor de base de dados, enquanto que, no não-relacional, desenvolvemos um componente de coprocessamento para estender as funcionalidades deste.

Após a análise do sistema desenvolvido, concluímos que a integração de algoritmos de *secret sharing*, com fortes garantias de privacidade, no modelo de segurança discutido, introduz uma degradação significativa de *performance*, quando comparado com sistemas de dados sem segurança.

As técnicas de *secret sharing* aplicadas requerem que algumas das operações sejam realizadas de forma distribuída, usando várias rondas de comunicação que, tendo um custo computacional elevado, aumentam a complexidade do sistema. Para amenizar esta degradação de *performance*, mostramos que, aplicando determinadas alterações aos algoritmos utilizados no *Sharemind*, obtemos algoritmos que beneficiam o desempenho em detrimento de algumas propriedades de segurança e privacidade. Um destes algoritmos permitiu reduzir consideravelmente o tempo de execução do sistema, aproximando o seu desempenho ao de um sistema de dados sem segurança.

Provamos, com recurso a provas de segurança, que a informação adicional libertada por estes

algoritmos, é limitada e, por isso, aplicável em casos em que o relaxamento das propriedades de segurança é uma opção. *Grosso modo*, estes algoritmos permitem, ao utilizador, ajustar a segurança do sistema conforme as necessidades de desempenho/privacidade.

Por fim, concluímos que, é possível integrar sistemas de dados relacionais e não-relacionais, em ambientes de computação segura, preservando a possibilidade de consultar os sistemas usando *SQL*. É, no entanto, necessário desenvolver algoritmos significativamente mais eficientes e implementar novas funcionalidades, para reduzir a distância entre os sistemas de dados seguros e os sistemas atuais, sem segurança.

5.1. Trabalho futuro

Um dos principais pontos a desenvolver, em trabalho futuro, passa por alargar consideravelmente o subconjunto de *queries*, suportadas pelo sistema. Implementar o suporte para operações tais como, *JOIN's*, *COUNT's*, *MAX/MIN's*, ou para outros tipos de dados, tais como *floats*, nas *queries*, é possível, usando novos algoritmos para dados *secret shared* [5, 6, 14, 16, 17]. Estas, possíveis, novas funcionalidades do sistema, garantiriam uma melhor e mais alargada usabilidade do sistema, aproximando-o dos sistemas de dados tradicionais, sem segurança.

Outro ponto de interesse seria o desenvolvimento de novos algoritmos, que operem sobre dados *secret shared*, que tenham um melhor desempenho que os atuais, mantendo as propriedades de segurança e privacidade apresentadas. Mais uma vez isto aproximaria o sistema apresentado dos sistemas de dados tradicionais, tendo em conta o desempenho geral do sistema.

Por fim, seria importante descobrir formas de contornar a questão da unicidade das chaves primárias, para permitir a execução de *index scans*, em vez de *table scans*, sobre dados *secret shared*. O que permitiria melhorar significativamente o desempenho dos *scans* no sistema e precisa, por isso, de ser abordado num trabalho futuro.

Todas estas novas funcionalidades e melhorias serviriam como mais valias para que estes sistemas de dados pudessem ganhar uma maior adoção e importância, no seio dos sistemas de armazenamento e computação de dados.

Bibliografia

[1] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The reactive simulatability (rsim) framework for asynchronous systems. *Inf. Comput.*, 205(12):1685–1720, December 2007. ISSN 0890-5401. doi: 10.1016/j.ic.2007.05.002. URL http://dx.doi.org/10.1016/j.ic.2007.05.002.

[2] G.R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, Monval, NJ, USA, 1979. AFIPS Press.

[3] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013. http://hdl.handle.net/10062/29041.

[4] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012. ISSN 1615-5262. http://dx.doi.org/10.1007/s10207-012-0177-2.

[5] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis (short paper). In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security. FC'12*, pages 57–64, 2012.

[6] Dan Bogdanov, Liina Kamm, Sven Laur, Pille Pruulmann-Vengerfeldt, Riivo Talviste, and Jan Willemson. Privacy-preserving statistical data analysis on federated databases. In *Proceedings of the Annual Privacy Forum. APF'14*, volume 8450 of *LNCS*, pages 30–55. Springer, 2014.

[7] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From input private to universally composable secure multiparty computation primitives. Cryptology ePrint Archive, Report 2014/201, 2014. http://eprint.iacr.org/.

[8] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. http://eprint.iacr.org/.

[9] Juan A. Garay, Philip MacKenzie, and Ke Yang. Efficient and secure multi-party computation with faulty majority and complete fairness. In *In Cryptology ePrint Archive, http://eprint.iacr.org/2004/019*, 2004.

[10] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 1 edition, 2011. ISBN 1449396100. URL http://www.amazon.de/HBase-Definitive-Guide-Lars-George/dp/1449396100/ref=sr_1_1?ie=UTF8&qid=1317281653&sr=8-1.

[11] Dennis Hofheinz and Victor Shoup. Gnuc: A new universal composability framework. Cryptology ePrint Archive, Report 2011/303, 2011. http://eprint.iacr.org/.

[12] Perry Hooker and Mike Rosulek. A very brief overview of the universally composable security framework, 2011.

[13] Roman Jagomägis. SecreC: a Privacy-Aware Programming Language with Applications in Data Mining. Master's thesis, Institute of Computer Science, University of Tartu, 2010.

[14] Liina Kamm and Jan Willemson. Secure floating-point arithmetic and private satellite collision analysis. Cryptology ePrint Archive, Report 2013/850, 2013. http://eprint.iacr.org/.

[15] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 29(7):886–893, 2013. URL http://bioinformatics.oxfordjournals.org/content/29/7/886.abstract.

[16] Toomas Krips and Jan Willemson. Hybrid Model of Fixed and Floating Point Numbers in Secure Multiparty Computations. Cryptology ePrint Archive, Report 2014/221, 2014. http://eprint.iacr.org/.

[17] Sven Laur, Riivo Talviste, and Jan Willemson. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *Applied Cryptography and Network Security*, volume 7954 of *LNCS*, pages 84–101. Springer, 2013.

[18] Martin Pettai and Peeter Laud. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. *IACR Cryptology ePrint Archive*, 2014:240, 2014.

[19] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043566. URL http://doi.acm.org/10.1145/2043556.2043566.

[20] Raluca Ada Popa, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: A practical encrypted relational dbms, 2011. URL http://people.csail.mit.edu/nickolai/papers/raluca-cryptdb-tr.pdf.

[21] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, September 2012. ISSN 0001-0782. doi: 10.1145/2330667.2330691. URL http://doi.acm.org/10.1145/2330667.2330691.

[22] Reimo Rebane. An integrated development environment for the SecreC programming language. Bachelor's thesis. University of Tartu, 2010.

[23] Jesper Buus Nielsen Ronald Cramer, Ivan Damgard. Multiparty computation, an introduction. 2009.

[24] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. ISSN 0001-0782. doi: 10.1145/359168.359176. http://doi.acm.org/10.1145/359168.359176.

[25] Ricardo Vilaça, Francisco Cruz, José Orlando Pereira, and Rui Oliveira. An effective scalable sql engine for nosql databases. In *Proceedings of the 13th IFIP Distributed Applications and Interoperable Systems (DAIS)*, LNCS, Florence, Italy, June 2013. IFIP, IFIP. doi: 10.1007/978-3-642-38541-4_12.

[26] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, 1982. http://dx.doi.org/10.1109/SFCS.1982.88], doi = 10.1109/SFCS.1982.88, acmid = 1382751, publisher = IEEE Computer Society, address = Washington, DC, USA,.

Apêndice A

Algoritmos de *Secret Sharing*

Algoritmo 10: Protocolo $[\![u]\!]$ Deshare(u_1, u_2, u_3) para recuperar u a partir das suas *shares*

Dados: *shares* u_1, u_2 e u_3

Resultado: Valor $[\![u]\!]$ tal que $u \equiv (u_1 + u_2 + u_3) \mod \mathbb{Z}_{2^n}$

1 Calcula $u = (u_1 + u_2 + u_3) \mod 2^n$

2 Retorna u

Algoritmo 11: Protocolo $[\![w]\!] \leftarrow$ Reshare$([\![u]\!])$ para *resharing*

Dados: Valor partilhado $[\![u]\!]$

Resultado: Valor $[\![w]\!]$ partilhado tal que w=u, com todas as *shares* w_i distribuídas uniformemente e u_i e w_i independentes para i,j= 1,2,3

1 $\mathcal{P}_1$ gera aleatoriamente $r_{12} \leftarrow \mathbb{Z}_{2^n}$

2 $\mathcal{P}_2$ gera aleatoriamente $r_{23} \leftarrow \mathbb{Z}_{2^n}$

3 $\mathcal{P}_3$ gera aleatoriamente $r_{31} \leftarrow \mathbb{Z}_{2^n}$

4 Todos os valores $*_{ij}$ são enviados de $\mathcal{P}_i$ para $\mathcal{P}_j$

5 $\mathcal{P}_1$ calcula $w_1 \leftarrow u_1 + r12 - r_{31}$

6 $\mathcal{P}_2$ calcula $w_2 \leftarrow u_2 + r23 - r_{12}$

7 $\mathcal{P}_3$ calcula $w_3 \leftarrow u_3 + r31 - r_{23}$

8 Retorna $[\![w]\!]$

Algoritmo 12: Protocolo $[\![u']\!] \leftarrow$ ReshareToTwo($[\![u]\!]$) para fazer reshare a um valor $[\![u]\!]$ entre $\mathcal{P}_2$ e $\mathcal{P}_3$

Dados: Valor partilhado $[\![u]\!]$

Resultado: Valor $[\![u']\!]$ partilhado tal que u = u' e $u'_1 = 0$

$\mathcal{P}_1$ gera $r_2 \leftarrow \mathbb{Z}_{2^n}$ aleatório e calcula $r_3 \leftarrow u_1 - r_2$

$\mathcal{P}_1$ define $u'_1 = 0$ e envia r_i a $\mathcal{P}_i$ (i=2,3)

$\mathcal{P}_i$ calcula $u'_i \leftarrow u_i + r_i$ (i=2,3)

Retorna $[\![u']\!]$

Algoritmo 13: $\overline{[\![p']\!]} \leftarrow$ PrefixOR($\overline{[\![p]\!]}$)

Dados: Vector $\overline{[\![p]\!]}$ partilhado de forma *bitwise*

Resultado: Vector $\overline{[\![p']\!]}$ com a forma 00...011...1, onde a parte inicial 00...01 coincide com o vector originalmente representado por $\overline{[\![p]\!]}$

$\ell \leftarrow |\overline{[\![p]\!]}|$

if ℓ = *1* **then**

  Retorna $\overline{[\![p']\!]} \leftarrow \overline{[\![p]\!]}$

else

  $\overline{[\![p']\!]}^{(\ell-1\ldots\lfloor \ell/2 \rfloor)} \leftarrow$ PrefixOR($\overline{[\![p]\!]}^{(\ell-1\ldots\lfloor \ell/2 \rfloor)}$)

  $\overline{[\![p']\!]}^{(\lfloor \ell/2 \rfloor-1\ldots0)} \leftarrow$ PrefixOR($\overline{[\![p]\!]}^{(\lfloor \ell/2 \rfloor-1\ldots0)}$)

  for $i \leftarrow 0$ ***to*** $\lfloor \ell/2 \rfloor$ - *1* **do**

    $\overline{[\![p']\!]}^{(i)} \leftarrow \overline{[\![p']\!]}^{(i)} \vee \overline{[\![p']\!]}^{(\lfloor \ell/2 \rfloor)}$

    Retorna $\overline{[\![p']\!]}$

Algoritmo 14: Protocolo $[\![s]\!] \leftarrow \text{MSNZB}(\overline{[\![u]\!]})$ para determinar a posição do bit não-zero mais significativo

Dados: Valor partilhado $[\![u]\!]$ de forma *bitwise*

Resultado: Vector partilhado $\overline{[\![s]\!]}$ tal que $\overline{[\![s]\!]}^{j}$ representa 1, onde j é a posição mais significativa, onde $\overline{[\![u]\!]}^{j}$ representa 1, e 0 caso contrário. Se todos os bits de $\overline{[\![u]\!]}$ resprenstam 0,todos os bits partilhados de $\overline{[\![s]\!]}$ representam 0 também

1 $[\![u']\!] \leftarrow PrefixOR(\overline{[\![u]\!]})$

2 **for** $i \leftarrow 0$ ***to*** n - 2 **do**

3 $\quad \overline{[\![s]\!]}^{(i)} \leftarrow \overline{[\![u']\!]}^{(i)} \oplus \overline{[\![u']\!]}^{(i+1)}$

4 $\overline{[\![s]\!]}^{(n-1)} \leftarrow \overline{[\![u']\!]}^{(n-1)}$

5 Retorna $\overline{[\![s]\!]}$

Algoritmo 15: Protocolo $[\![\lambda]\!] \leftarrow \text{Overflow}([\![u']\!])$ para obter o bit de overflow $[\![\lambda]\!]$ para $[\![u']\!]$ se a *share* $u'_1 = 0$

Dados: Valor partilhado $[\![u]\!]$ com $u'_1 = 0$

Resultado: Valor $[\![\lambda]\!]$ partilhado tal que $u' = u'_2 + u'_3 - \lambda 2^n$

1 $\mathcal{P}_1$ define $p_1 = 0$

2 $\mathcal{P}_2$ define $p_2 = u'_2$

3 $\mathcal{P}_3$ define $p_3 = -u'_3$

4 $\overline{[\![s]\!]} \leftarrow MSNZB(\overline{[\![p]\!]})$

5 $\mathcal{P}_3$ partilha o valor $-u'3$ *bitwise* como o vector $\overline{[\![-u'_3]\!]}$

6 $[\![\lambda^0]\!] \leftarrow 1 \oplus \bigoplus_{i=0}^{n-1} \overline{[\![s]\!]}^{(i)} \wedge \overline{[\![-u'_3]\!]}^{(i)}$

7 $\mathcal{P}_3$ verifica se $u'_3 = 0$. Se tal, $\lambda_3^0 = 1 \oplus \lambda_3^0$

8 Retorna $[\![\lambda]\!] \leftarrow ShareConv[\![\lambda^0]\!]$

Algoritmo 16: Protocolo $[\![v]\!] \leftarrow$ ShareConv($[\![u]\!]$) para converter uma *share* $[\![u]\!] \in \mathbb{Z}_2$ para $[\![v]\!] \in \mathbb{Z}_{2^n}$

Dados: Valor partilhado $[\![u]\!]$ em *shares* de bits

Resultado: Valor partilhado $[\![v]\!]$ tal que u = v, e $[\![v]\!]$ é partilhado em $\mathbb{Z}_{2^n}$

1 $\mathcal{P}_1$ gera aleatoriamente b $\leftarrow \mathbb{Z}_2$ e define m $\leftarrow$ b $\oplus u_1$

2 $\mathcal{P}_1$ converte localmente m para $\mathbb{Z}_{2^n}$, e gera aleatoriamente $m_{12} \leftarrow \mathbb{Z}_{2^n}$ e calcula $m_{13} = m - m_{12}$

3 $\mathcal{P}_1$ gera aleatoriamente $b_{12} \leftarrow \mathbb{Z}_2$ e calcula $b_{13} = b - b_{12} = b \oplus b_{12}$

4 Todos os valores $*_{ij}$ são enviados de $\mathcal{P}_i$ para $\mathcal{P}_j$

5 $\mathcal{P}_2$ gera aleatoriamente $r_{23} \leftarrow \mathbb{Z}_{2^n}$

6 $\mathcal{P}_3$ gera aleatoriamente $r_{31} \leftarrow \mathbb{Z}_{2^n}$

7 Todos os valores $*_{ij}$ são enviados de $\mathcal{P}_i$ para $\mathcal{P}_j$

8 $\mathcal{P}_1$ define $v_1 \leftarrow 0$

9 $\mathcal{P}_2$ e $\mathcal{P}_3$ definem $s \leftarrow s_{23} \oplus s_{32}$

10 **if** $s = 1$ **then**

11 $\quad$ $\mathcal{P}_2$ define $v_2 \leftarrow (1 - m_{12})$

12 $\quad$ $\mathcal{P}_3$ define $v_3 \leftarrow (-m_{13})$

13 **else**

14 $\quad$ $\mathcal{P}_2$ define $v_2 \leftarrow m_{12}$

15 $\quad$ $\mathcal{P}_3$ define $v_3 \leftarrow m_{13}$

16 Retorna $[\![v]\!]$

Algoritmo 17: Protocolo $[\![w]\!] \leftarrow$ ShiftR($[\![u]\!]$, p) para avaliar o shift à direita

Dados: Valores partilhado $[\![u]\!]$ e um valor público de shift p

Resultado: Valor $[\![w]\!]$ partilhado tal que w = u $\gg$ p

1 $[\![u']\!] \leftarrow ReshareToTwo([\![u]\!])$

2 $[\![s]\!] \leftarrow [\![u' \ll n-p]\!]$ (localmente)

3 $[\![\lambda_1]\!] \leftarrow Overflow(u')$

4 $[\![\lambda_2]\!] \leftarrow Overflow(s)$

5 $\mathcal{P}_i$ calcula $v_i \leftarrow u'_i \gg p$

6 Retorna $[\![w]\!] = [\![v]\!] - 2^{n-p}[\![\lambda_1]\!] + [\![\lambda_2]\!]$

Algoritmo 18: Protocolo $[\![w]\!] \leftarrow$ BitConj($\overline{[\![b]\!]}$) para determinar a conjunção dos bits de b

Dados: Vector partilhado $\overline{[\![b]\!]}$ de bits

Resultado: Valor $[\![w]\!]$ partilhado tal que w é a conjunção dos bits de $\overline{[\![b]\!]}$

1 $w \leftarrow \overline{[\![b]\!]}^{(0)}$

2 **for** $i = 1$ ***to*** n **do**

3 $\quad w =$Mult$(w, \overline{[\![b]\!]}^{(i)})$

4 Retorna $[\![w]\!]$