Pontifícia Universidade Católica
do Rio de Janeiro

Guilherme Gomes Felix da Silva

PUC-Rio - Certificação Digital Nº 1521382/CA

Formalização de Algoritmos de Criptografia em um Assistente de Provas Interativo

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre em Informática pelo Programa de Pós-Graduação da PUC-Rio.

Orientador: Prof. Edward Hermann Haeusler

Rio de Janeiro
agosto de 2018

Guilherme Gomes Felix da Silva

Formalização de Algoritmos de Criptografia em um Assistente de Provas Interativo

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Edward Hermann Haeusler
Orientador
Departamento de Informática – PUC-Rio

Markus Endler
Departamento de Informática – PUC-Rio

Christiano de Oliveira Braga
UFF

Jefferson de Barros Santos
FGV

Prof. Márcio da Silveira Carvalho
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 28 de agosto de 2018

PUC-Rio - Certificação Digital Nº 1521382/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Guilherme Gomes Felix da Silva

Formou-se em Engenharia da Computação pela Pontifícia Universidade Católica do Rio de Janeiro em 2015.

Ficha Catalográfica

Silva, Guilherme Gomes Felix da

Formalização de algoritmos de criptografia em um assistente de provas interativo / Guilherme Gomes Felix da Silva ; orientador: Edward Hermann Haeusler. – 2018.

70 f. : il. ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2018.

Inclui bibliografia

1. Informática – Teses. 2. Criptografia. 3. Assistente de provas. I. Haeusler, Edward Hermann. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

PUC-Rio - Certificação Digital Nº 1521382/CA

Agradecimentos

Ao meu orientador, Edward Hermann Haeusler, pela constante ajuda e orientação nas pesquisas feitas durante a realização deste trabalho.

À CAPES e à PUC-Rio pelo auxílio financeiro sem o qual este trabalho não poderia ter sido realizado.

À minha família, pelo apoio dado todos os dias.

PUC-Rio - Certificação Digital Nº 1521382/CA

Resumo

Silva, Guilherme Gomes Felix da; Haeusler, Edward Hermann. **Formalização de Algoritmos de Criptografia em um Assistente de Provas Interativo**. Rio de Janeiro, 2018. 70p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Ao descrever-se a prova de um teorema, é fundamental que haja cautela para que esta não contenha erros ou inconsistências. Para provas muito longas, no entanto, a detecção de erros pode tornar-se uma tarefa humanamente inviável. Um assistente de provas é um programa cuja finalidade é realizar esta detecção de erros para um usuário de forma eficiente, bem como facilitar a construção e compreensão de provas complexas a partir de outras já existentes. O Lean Theorem Prover, desenvolvido em 2012 por Leonardo de Moura, é um assistente de provas que trabalha com descrição de provas através de uma linguagem computacional compilável. Propomos aqui uma descrição no Lean Theorem Prover das provas de funcionamento de diversos algoritmos pertinentes à área de criptografia.

Palavras-chave

Criptografia; assistente de provas.

PUC-Rio - Certificação Digital Nº 1521382/CA

Abstract

Silva, Guilherme Gomes Felix da; Haeusler, Edward Hermann (Advisor). **Formalization of Cryptography Algorithms in an Interactive Theorem Prover**. Rio de Janeiro, 2018. 70p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

When describing a proof of a theorem, one must be cautious to ensure said proof does not contain errors or inconsistencies. For very long proofs, however, error detection can become humanly infeasible. A proof assistant is a program whose purpose is to perform said error detection efficiently, as well as to assist in the creation and comprehension of complex proofs out of simpler, existing proofs. The Lean Theorem Prover, developed in 2012 by Leonardo de Moura, is a proof assistant which functions via description of proofs in a compilable computer language. We present a description of proofs of correctness of various algorithms pertaining to cryptography in the Lean Theorem Prover.

Keywords

Cryptography; proof assistant.

PUC-Rio - Certificação Digital Nº 1521382/CA

Sumário

PUC-Rio - Certificação Digital Nº 1521382/CA

Lista de figuras

PUC-Rio - Certificação Digital Nº 1521382/CA

Lista de tabelas

PUC-Rio - Certificação Digital Nº 1521382/CA

1 Introdução

Ao descrever-se uma prova de um teorema, em particular provas de tamanho extenso, a detecção de erros torna-se uma questão crucial, pois um único erro pode implicar em uma conclusão errônea sobre a veracidade ou falsidade de um teorema. De modo a compensar a imperfeição humana no que se diz respeito à detecção de erros em provas de teoremas, foram desenvolvidos programas que visam auxiliar nesta prática, denominados assistentes de provas.

O trabalho aqui apresentado tem a intenção de explorar o uso de assistentes de provas interativos, utilizando um destes programas para demonstrar a corretude de provas relacionadas a algoritmos da área de criptografia.

Este trabalho encontra-se dividido em diversas partes. A seção 2 deste documento tem a finalidade de apresentar os conceitos básicos de criptografia que são necessários para a compreensão do trabalho desenvolvido, como a diferença entre criptografia com chave simétrica e assimétrica e o conceito de chaves públicas e privadas.

Na seção 3, são apresentados em maior detalhe os algoritmos de criptografia explorados neste trabalho (*Data Encryption Standard*, RSA e Blowfish). Aqui também são descritos em maior detalhe alguns dos conceitos apresentados na seção 2, conforme estes tornam-se relevantes para a compreensão dos algoritmos. Como o algoritmo RSA utiliza aritmética modular amplamente, está incluída uma seção que visa apresentar o leitor que não possua familiaridade a esta área da matemática.

A seção 4 visa descrever o Lean Theorem Prover, que é o assistente de provas interativo utilizado para o desenvolvimento das provas formais aqui apresentadas.

Por fim, a seção 5 descreve o trabalho desenvolvido propriamente dito – isto é, como as provas de corretude dos algoritmos previamente descritos foram traduzidas para a linguagem do Lean Theorem Prover.

1.1 Contribuições

Como contribuição para a comunidade científica, visa-se explorar aqui o conceito de prova interativa de teoremas e aplicar estas provas à área de criptografia, por tratar-se de uma área na qual foram realizados relativamente poucos trabalhos envolvendo provas formais e interativas, e por não ser conhecido, até o fechamento deste trabalho, nenhum que houvesse utilizado o Lean Theorem Prover para tal fim.

PUC-Rio - Certificação Digital Nº 1521382/CA

2 Conceitos Fundamentais de Criptografia

Criptografia é a ciência que estuda meios de garantir a segurança na comunicação entre dois participantes de um diálogo através de codificação de mensagens. Codificar uma mensagem significa alterá-la por meio de um processo ou algoritmo que utilize um determinado segredo, ou *chave*, de modo que o conteúdo da mensagem se torne indecifrável para um observador, sendo possível apenas para um indivíduo que possua a chave recuperar a mensagem original. Uma vez codificada, uma mensagem pode ser transferida entre dois interlocutores através de um *meio inseguro*, no qual esteja sujeita a ser observada por um terceiro indivíduo, ou *adversário*.

Define-se aqui como adversário qualquer indivíduo para o qual a mensagem não tenha sido intencionada. Caso este venha a interceptar a mensagem, não conseguirá discernir o seu significado sem posse da chave e verá apenas uma sequência incompreensível de dados.

Idealmente, um algoritmo de criptografia deve garantir duas coisas:

- Que uma mensagem possa ser codificada ou decodificada de forma relativamente rápida ou eficiente com posse da chave, e
- Que seja impossível ou inviável decodificar uma mensagem sem posse da chave utilizada para codificá-la.

Embora algoritmos de criptografia sejam utilizados há mais de 2000 anos, são relevantes para este trabalho apenas aqueles desenvolvidos a partir da década de 1970 e utilizados amplamente no contexto de informática. Neste contexto, um algoritmo de codificação consiste de uma função $f(x, k) = y$, onde x é o dado original, k é a chave, e y é o dado codificado, ou *cifra*. Aqui, x, k e y consistem, em seu nível mais baixo, de sequências de bits. Dependendo do algoritmo utilizado, no entanto, estas sequências podem ser tratadas como uma representação de algo em um nível de abstração mais alto – algoritmos de chave assimétrica, em particular, tratam estes dados como números naturais ou inteiros, sendo baseados em operações de aritmética modular sobre estes valores.

O comprimento de uma chave também varia com o algoritmo. Os dois algoritmos de chave simétrica mais amplamente reconhecidos utilizam chaves relativamente curtas; o *Data Encryption Standard* (DES), trabalha com chaves de 56 bits, ao passo que o *Advanced Encryption Standard* (AES) utiliza chaves que podem variar de 128 a 256 bits. Algoritmos de chave assimétrica costumam utilizar chaves maiores – O RSA, um algoritmo com chave assimétrica que utiliza funções de codificação baseadas em fatoração de primos, trabalha com chaves da ordem de milhares de bits, geralmente entre 1024 e 4096.

Outra distinção importante que deve ser feita antes de nos aprofundarmos em conceitos de criptografia diz respeito a algoritmos de criptografia com chave de bloco

PUC-Rio - Certificação Digital Nº 1521382/CA

(*block cipher*) ou chave de fluxo (*stream cipher*). Em um algoritmo com chave de bloco, o dado x é tratado pela operação de codificação como um conjunto de blocos de tamanhos fixos e idênticos, sendo cada um destes blocos codificado de forma independente dos demais; em seguida, os blocos codificados são concatenados de modo a formar a cifra y. Os algoritmos com chave de fluxo, por outro lado, codificam os bits de entrada de forma contínua, sendo o valor de cada bit da cifra resultante influenciado por aqueles que o precederam. Foram utilizados para os fins deste trabalho apenas algoritmos com chave de bloco.

2.1 Criptografia com Chave Simétrica

Em um algoritmo de criptografia com chave simétrica, a mesma chave é utilizada tanto para codificação de dados (pelo emissor) quanto para decodificação (pelo receptor). Para qualquer função de codificação f utilizada por um destes algoritmos, existe uma função inversa f^{-1} que é capaz de recuperar o dado original x atráves da cifra y e da chave k utilizada para codificação.

Assim, um usuário A que deseje enviar uma mensagem x para um outro usuário B aplica f para cifrar a mensagem, obtendo y. Em seguida, o dado cifrado y é enviado ao usuário B, que aplica f^{-1} com a mesma chave k para recuperar x. Cada chave está, portanto, associada a um diálogo entre um par de usuários. (Por motivos de segurança, uma chave deve ser descartada após um único uso, sendo necessária a geração de uma nova chave para qualquer diálogo subsequente). A figura 2.1 providencia uma representação genérica de um diálogo utilizando um algoritmo de chave simétrica.

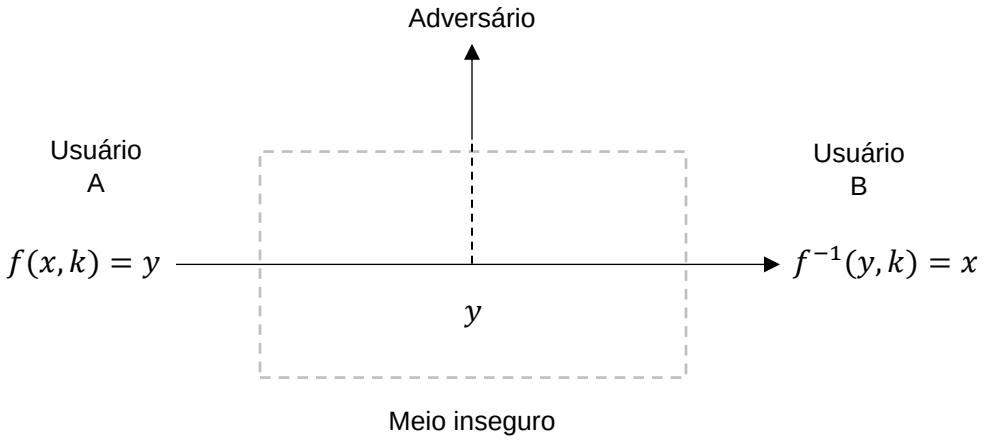

Fig 2.1: Representação genérica do funcionamento de um algoritmo de criptografia com chave simétrica

PUC-Rio - Certificação Digital Nº 1521382/CA

Como se pode ver, o diálogo ilustrado na figura 2.1 está sendo espionado por um adversário. Isto não é uma situação de exceção; um princípio fundamental da boa criptografia dita que todo e qualquer algoritmo deve ser desenvolvido sob uma mentalidade "paranoica" – isto é, partindo do pressuposto que não só todo diálogo estará sendo espionado por um adversário, mas que este se valerá de todos os meios possíveis para tentar deduzir a chave e assim "quebrar" a cifra y. (A prática de desenvolver meios de quebrar cifras é denominada *criptoanálise* e se trata de uma área de estudo intrinsecamente ligada à criptografia, porém, não é um foco deste trabalho.)

Assumimos, portanto, que o adversário sempre tem acesso à cifra y. Ele também tem acesso a f, pois é necessário, por questões de segurança, que um bom algoritmo de criptografia seja de conhecimento público. Nossa única garantia de segurança, portanto, é que o adversário não possa recuperar x sem posse da chave k. Garantir isto é o papel de um algoritmo de criptografia, ao passo que cabe aos protocolos impedir que o adversário encontre uma maneira de obter k.

No entanto, deparamo-nos aqui com um problema, denominado o problema de *distribuição de chave* (*key establishment*): para que um algoritmo de criptografia com chave pública funcione, é necessário que tanto o receptor quanto o emissor tenham conhecimento da mesma chave k. Portanto, em algum momento, deve ser realizado o compartilhamento desta entre eles. No entanto, o canal de comunicação entre emissor e receptor obviamente não é seguro. Como uma chave nada mais é que um dado representado por uma sequência de bits, uma sugestão ingênua poderia ser que ela própria fosse codificada antes de realizada a transmissão, mas, para isto, seria necessária outra chave que teria de ser compartilhada através de um meio inseguro, e assim por diante. Outra sugestão poderia ser que os usuários se encontrassem em pessoa para realizar o compartilhamento da chave através de um meio seguro, porém, isto raramente é viável pelo fato de que, como já mencionado, uma chave é descartável e válida apenas pela duração de um diálogo.

Além disto, um segundo problema associado à criptografia com chave simétrica diz respeito à *complexidade* do protocolo utilizado. Como é necessário que uma chave exista para cada par emissor/receptor possível, o número de chaves é uma função quadrática, ou $O(n^2)$, do número de usuários ou participantes em um ambiente. Isto resulta em uma grande quantidade de chaves cujo armazenamento e transmissão pode se tornar ineficiente ou inviável para ambientes com um grande número de usuários.

Claramente, os algoritmos de criptografia com chave simétrica, embora extremamente importantes, não são o suficiente para cobrir todas as necessidades de segurança. Uma das possíveis soluções para ambos estes problemas está nos protocolos de *criptografia com chave pública ou assimétrica*, descritos a seguir.

PUC-Rio - Certificação Digital Nº 1521382/CA

2.2 Criptografia com Chave Pública (ou Chave Assimétrica)

O conceito de criptografia com chave pública ou assimétrica é uma invenção muito mais recente do que a criptografia simétrica, tendo sido proposto em 1976 por Whitfield Diffie e Martin Hellman. Em um algoritmo de criptografia com chave pública ou assimétrica, ao invés de ser definida uma chave para cada par emissor/receptor, cada usuário possui um par de chaves: uma chave pública, utilizada para codificação de dados, e uma chave privada, utilizada para decodificação.

A ideia por trás destes algoritmos é que cada usuário disponibilize sua chave pública para quaisquer outros com os quais possa vir a ter intenção de comunicar-se, e mantenha sua chave privada em segredo. Quem codifica uma mensagem antes de enviá-la não o faz com sua própria chave pública, mas com aquela pertencente ao receptor da mensagem, que, em seguida, realiza a decodificação com sua chave privada e recupera o dado original. A chave pública é enviada através de um canal inseguro e, portanto, pode ser interceptada por um adversário, mas isto não apresenta um problema de segurança, pois ela não pode ser usada para decodificação de dados.

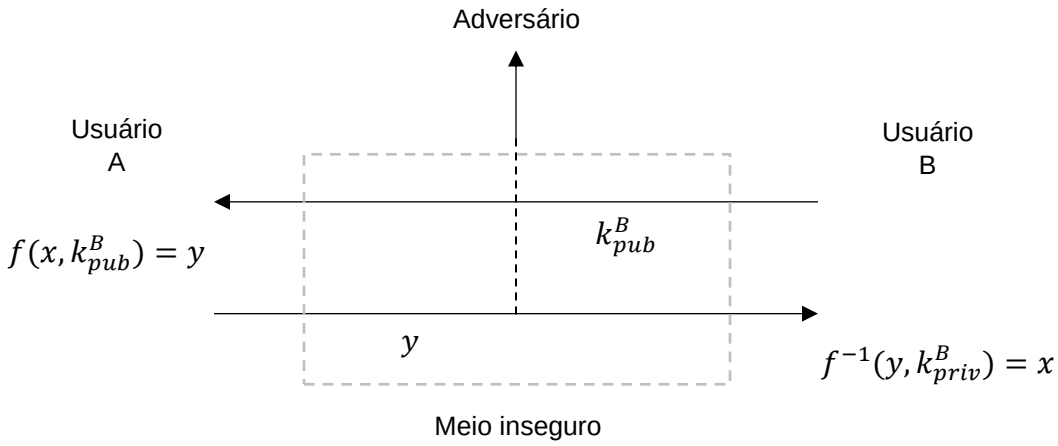

Fig 2.2: Representação genérica do funcionamento de um algoritmo de criptografia com chave pública

A relação entre as chaves pública e privada não é arbitrária; de fato, para que a chave privada seja capaz de decodificar uma mensagem codificada com a chave pública, é necessário que elas sejam geradas como um par. Para que as duas chaves funcionem como tal sem que a chave privada possa ser deduzida a partir da pública, um algoritmo de criptografia com chave pública deve ser baseado em uma função de mão única (*one-way function*) – isto é, uma função f para a qual seja fácil calcular $f(x) = y$, mas computacionalmente inviável obter x conhecendo y e f. Um exemplo de tal função envolve o problema de fatoração de primos mencionado anteriormente: dados dois números primos p e q suficientemente grandes, é viável calcular $n = p \cdot q$, mas inviável determinar p e q conhecendo apenas n. Este exemplo é utilizado pelo

PUC-Rio - Certificação Digital Nº 1521382/CA

algoritmo de criptografia com chave pública RSA (Rivest, Shamir e Adleman), descrito na seção 3.2 deste trabalho.

Embora estes algoritmos sejam menos eficientes do que um algoritmo com chave simétrica devido ao tamanho das chaves, é fácil ver como um algoritmo com chave pública resolve a questão de distribuição de chave: basta usar um algoritmo com chave pública para realizar a transmissão segura de uma chave simétrica para um algoritmo como o DES. Uma vez estabelecida, esta chave simétrica pode ser utilizada para transmissão. De fato, é extremamente comum que algoritmos com chave simétrica e assimétrica se complementem desta maneira. Um algoritmo assimétrico também providencia uma possível solução para o problema de complexidade mencionado anteriormente: por haver apenas um par de chaves por usuário, a quantidade de chaves existentes é uma função linear do número de participantes em um ambiente.

PUC-Rio - Certificação Digital Nº 1521382/CA

3 Algoritmos de Criptografia

3.1 O *Data Encryption Standard* (DES)

O *Data Encryption Standard*, ou DES, é um algoritmo de criptografia com chave simétrica inicialmente proposto na década de 1970 e aprovado em 1976 pelo Instituto Nacional de Padrões e Tecnologia dos Estados Unidos (NBS). Trata-se de um algoritmo de cifra de bloco que codifica blocos de 64 bits por vez, utilizando uma chave de 56 bits para codificação. Note que, embora o tamanho de chave utilizado pelo DES tecnicamente seja de 64 bits (e o algoritmo seja comumente descrito como utilizando chaves deste tamanho), 8 destes 64 bits são bits de paridade que dependem dos demais 56 e são ignorados pelo algoritmo para fins de codificação. Na prática, portanto, o DES trata-se de um algoritmo com chave de 56 bits.

Embora tenha sido o padrão de criptografia com cifra de bloco mais utilizado por cerca de 30 anos, o DES não é mais considerado completamente seguro atualmente, tendo sido em grande parte substituído pelo AES. Isto se deve ao fato de que o tamanho relativamente pequeno da chave do DES torna possível quebrar uma cifra através de testes de força bruta (isto é, experimentando todas as chaves possíveis), embora este fato não tenha sido descoberto até a década de 1990.

PUC-Rio - Certificação Digital Nº 1521382/CA

3.1.1 Estrutura do Algoritmo

O algoritmo de codificação do DES consiste principalmente de 16 aplicações consecutivas de uma função de codificação, como ilustrado na figura 3.1. Note que, embora as 16 etapas de codificação sejam funcionalmente iguais, cada uma delas utiliza uma dentre 16 subchaves diferentes $k_1 \ldots k_{16}$. Estas subchaves possuem 48 bits e são derivadas da chave principal k através do algoritmo *Key Schedule*, descrito na seção 3.1.3.

São realizadas também duas operações de reposicionamento de bits denominadas permutação inicial e permutação final, que, respectivamente, precedem e sucedem as etapas de codificação. Estas operações não dependem da chave utilizada, tampouco possuem relevância criptográfica ou contribuem para a segurança do protocolo. Não se sabe com exatidão a razão pela qual estas operações de permutação foram criadas, porém, é possível que tenha sido para facilitar a obtenção de dados em barramentos de 8 bits.[2]

O DES utiliza a mesma função para codificação e decodificação; portanto, para recuperar-se o dado original x, basta aplicar-se novamente o algoritmo ilustrado abaixo, tendo como entrada o dado cifrado y e a chave k. Para que esta decodificação funcione corretamente, no entanto, é necessário que a operação de decodificação utilize as subchaves na ordem inversa daquela utilizada na codificação. Garantir que as chaves

sejam geradas em ordem inversa para a decodificação é responsabilidade do Key Schedule.

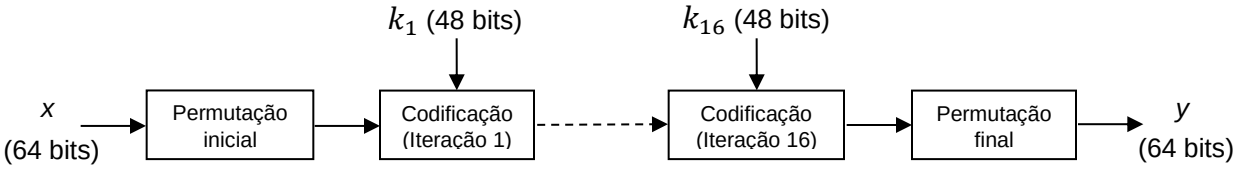

Fig 3.1: Estrutura do *Data Encryption Standard*

3.1.1.1 Etapas de Codificação

A figura 3.2 ilustra em detalhe uma das 16 etapas de codificação do DES. Como se pode ver, cada etapa altera apenas a metade esquerda do bloco de dados, isto é, os 32 bits mais significativos. Em seguida, as duas metades do dado têm suas posições trocadas para que seja realizada a codificação da outra metade na iteração seguinte. A função *f* abaixo é descrita na seção 3.1.2.

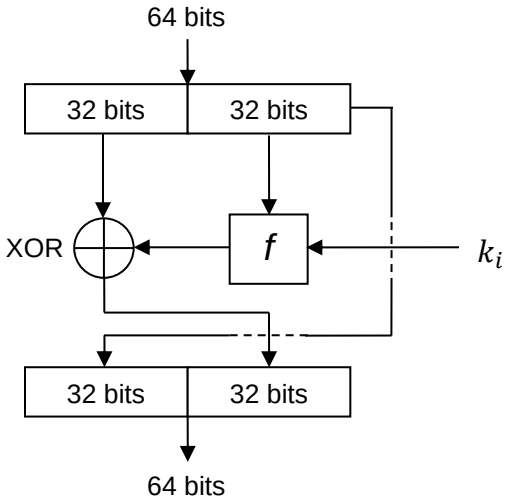

Fig 3.2: Estrutura de uma etapa de codificação do *Data Encryption Standard*

3.1.1.2 Permutações Inicial e Final

As etapas de permutação inicial e final, realizadas no início e no fim de uma aplicação do DES, consistem no reposicionamento dos 64 bits de entrada através de uma tabela.

PUC-Rio - Certificação Digital Nº 1521382/CA

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Tabelas 3.1 e 3.2: Permutações inicial (esquerda) e final (direita) do *Data Encryption Standard*

Como demonstração, vamos realizar um exemplo de aplicação deste algoritmo de permutação sobre a sequência de 64 bits abaixo, correspondente à sequência hexadecimal E5 5A 17 9F B7 64 EA 12.

11100101 01011010 00010111 10011111 10110111 01100100 11101010 00010010

A tabela abaixo ilustra o posicionamento passo-a-passo dos bits após a permutação, em intervalos de 8 bits e ordem crescente de posição. Por simplicidade, começamos posicionando os 8 primeiros bits da sequência, ou 11100101, seguidos pelo próximo intervalo de 8 bits, e assim por diante até que tenhamos determinado a posição de todos os 64 bits.

Note que este posicionamento é realizado da direita para a esquerda em cada intervalo de 8 bits da sequência resultante, conforme a organização dos valores na tabela 3.1. Cada linha da sequência original é transposta para uma coluna da sequência pós-permutação.

Posição	Dados	
1-8	11100101	_____**1** _____**0** _____**1** _____**1** _____**1** _____**1** _____**0** _____**0**
9-16	01011010	_____**1**1 _____**1**0 _____**0**1 _____**0**1 _____**0**1 _____**0**1 _____**1**0 _____**1**0
17-24	00010111	_____**0**11 _____**1**10 _____**1**01 _____**1**01 _____**0**01 _____**0**01 _____**0**10 _____**1**10
25-32	10011111	____**0**011 ____**1**110 ____**1**101 ____**1**101 ____**1**001 ____**0**001 ____**1**010 ____**1**110
33-40	10110111	___**0**0011 ___**1**1110 ___**1**1101 ___**1**1101 ___**1**1001 ___**1**0001 ___**0**1010 ___**1**1110
41-48	01100100	__**1**00011 __**0**11110 __**1**1101 __**0**11101 __**0**1001 __**1**10001 __**0**01010 __**0**11110
49-56	11101010	_**1**100011 _**0**011110 _**0**111101 _**0**011101 _**1**011001 _**1**110001 _**1**001010 _**1**011110
57-64	00010010	**0**1100011 **1**0011110 **0**0111101 **0**0011101 **0**1011001 **0**1110001 **0**1001010 **1**1011110

Tabela 3.3: Exemplo de funcionamento da operação de permutação inicial do *Data Encryption Standard*

PUC-Rio - Certificação Digital Nº 1521382/CA

Portanto, determinamos que a sequência de bits obtida após a permutação é

01100011 10011110 00111101 00011101 01011001 01110001 01001010 11011110

ou 63 9E 3D 1D 59 71 4A DE em representação hexadecimal.

Uma aplicação da permutação final produzirá novamente a sequência inicial E5 5A 17 9F B7 64 EA 12, demonstrando que a permutação final se trata da operação inversa da permutação inicial. Voltaremos a falar sobre as permutações inicial e final na seção 5.1.1, onde é descrita a prova de corretude de ambas realizada no Lean Theorem Prover para este trabalho.

3.1.2 Função *f*

A função *f* (abreviação de *Feistel*, em homenagem a Horst Feistel) é uma função de codificação de dados e um dos componentes do DES. Conforme demonstrado na figura 3.2, a função f é realizada uma vez em cada um dos 16 passos de codificação do algoritmo, tendo como entrada uma subchave k_i de 48 bits (sendo esta diferente em cada uma das 16 iterações) e um dado de 32 bits. A função *f* é o núcleo do DES e a fonte de toda a segurança criptográfica do algoritmo, não só pelo fato de ser a única parte deste diretamente influenciada pela chave, mas pelo fato de ser o único componente do DES com a propriedade de difusão, isto é, a propriedade de que a alteração de um único bit do valor de entrada afeta mais de um bit do valor de saída.

PUC-Rio - Certificação Digital Nº 1521382/CA

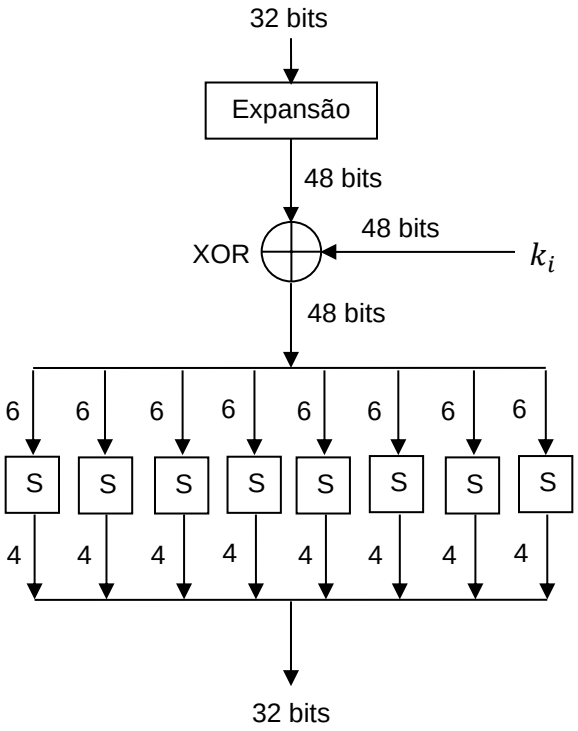

Fig 3.3: Estrutura da função *f* do *Data Encryption Standard*

3.1.3 *Key Schedule*

O algoritmo *Key Schedule* é utilizado para geração das 16 chaves utilizadas nas etapas individuais de codificação do DES, conforme ilustrado na figura 3.1. O primeiro passo do *Key Schedule* é uma operação de permutação de bits, denominada PC1 e realizada por meio de uma tabela, semelhante às operações de permutação inicial e final do DES. Como mencionado anteriormente, esta permutação ignora os 8 bits de paridade da chave k, mapeando-a em uma sequência de 56 bits. Em seguida, são realizadas sobre esta sequência de bits operações de deslocamento de bits para a esquerda, ou *shift-left*, sendo estas seguidas por outra permutação PC2 que mapeia as sequências resultantes em outras mais curtas, de 48 bits. São obtidas ao todo 16 sequências distintas de 48 bits, sendo estas as subchaves $k_1 \dots k_{16}$ utilizadas para codificação nas 16 etapas do DES. A ordem e disposição destas operações é ilustrada na figura 3.4 abaixo.

É importante mencionar que, além de o DES utilizar a mesma chave k para codificação e decodificação, as subchaves $k_1 \dots k_{16}$ também são as mesmas para ambas as operações. No entanto, a operação de decodificação utiliza-as em ordem inversa, iniciando com a subchave k_{16} e prosseguindo em ordem decrescente até k_1. De forma a aumentar a eficiência da operação de decodificação, é possível obter as subchaves em ordem decrescente a partir de k, aplicando operações de deslocamento à direita, ou *shift-right*.

PUC-Rio - Certificação Digital Nº 1521382/CA

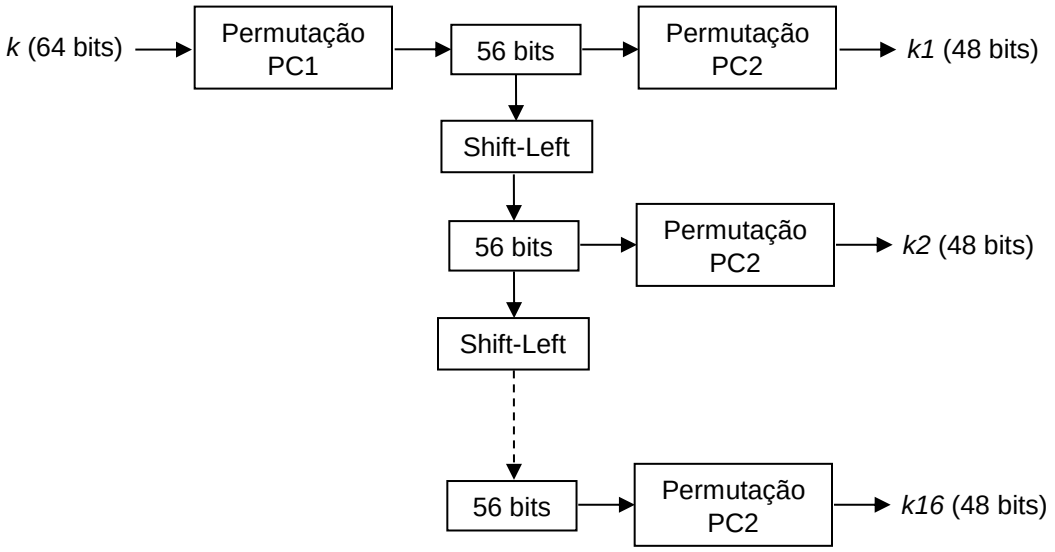

Fig 3.4: Estrutura da função *Key Schedule* do *Data Encryption Standard*

PUC-Rio - Certificação Digital Nº 1521382/CA

As provas de funcionamento do DES no Lean Theorem Prover são descritas na seção 5.1.

3.2 O Algoritmo RSA (Rivest, Shamir & Adleman)

O RSA é um algoritmo de criptografia com chave pública desenvolvido por Ron Rivest, Adi Shamir e Leonard Adleman, e apresentado pela primeira vez em 1977. Por ser um algoritmo com chave pública, cada execução do RSA por um usuário gera duas chaves diferentes que funcionam como um par: uma pública e uma privada, sendo a primeira destas a única a ser divulgada para outros usuários. Apenas a chave privada, mantida em segredo pelo usuário em questão, é capaz de decodificar as mensagens criptografadas com a sua chave pública e a ele enviadas.

Conforme mencionado na seção 2.2, o RSA é um algoritmo baseado no problema de fatoração de primos. Ele se vale do fato de que, até onde se sabe na comunidade científica atualmente, é computacionalmente muito mais simples calcular o produto de dois primos, $n = p \cdot q$, do que realizar a operação inversa – isto é, determinar p e q conhecendo-se apenas n. De fato, ao escolhermos valores suficientemente grandes para p e q, torna-se computacionalmente inviável realizar esta operação. Para este fim, o RSA utiliza valores para p e q com comprimento entre 512 e 2048 bits, o que resulta em um número n da ordem de 1024 a 4096 bits. O algoritmo trabalha, então, com números da ordem de 2^{512} ou maiores, ou seja, números com centenas de casas decimais.

O cálculo de n é, portanto, a função de mão única que serve como base para o estabelecimento de um par de chaves pública e privada para uma instância de

comunicação com RSA. Podemos ver p e q como os parâmetros de entrada desta função – são eles os valores mantidos em segredo e utilizados para cálculo da chave privada.

3.2.1 Conceitos de Aritmética Modular

A aritmética modular é utilizada extensivamente nas operações do RSA. Pretendemos aqui apresentar o mínimo necessário em relação a esta área da matemática para o acompanhamento do algoritmo, de modo que o leitor que já possui familiaridade com aritmética modular pode ignorar esta seção.

A aritmética modular lida com operações sobre números inteiros delimitadas por um valor conhecido como o módulo, que denominaremos m. O módulo m pode possuir qualquer valor natural maior do que 0. Ao realizarmos operações sob um módulo m (também conhecidas como operações em um *corpo finito* de tamanho m), trabalhamos com os inteiros $\{0, 1 \ldots m - 1\}$. Qualquer número fora deste espectro pode ser tratado como o resto da sua divisão por m. Desta forma,

$$23 \; mod \; 7 = 2,$$

pois o resto da divisão de 23 por 7 é igual a 2. Podemos aplicar a operação mod da mesma forma sobre qualquer número natural; no caso de $mod\ 7$, o resultado será sempre um dentre os valores $\{0, 1, 2, 3, 4, 5, 6\}$.

Dois números inteiros que possuam o mesmo resto quando divididos por m são ditos *congruentes* no módulo m, como no seguinte exemplo:

$$23 \equiv 30 \; (mod \; 7)$$

O conceito de congruência aqui definido também pode ser visto como a afirmação de que a diferença entre dois números a e b é um múltiplo de m, ou seja,

$$a \equiv b \; (mod \; m) \text{ se e somente se } \exists n \in \mathbb{Z}, a = b + n \cdot m.$$

Assim, podemos inclusive definir congruência envolvendo números negativos, como, por exemplo,

$$-1 \equiv 6 \; (mod \; 7),$$

pelo fato de a diferença entre estes números ser um múltiplo do módulo.

Como convenção de notação, é utilizado em todo este trabalho $mod\ m$, sem parênteses, para representar a operação aritmética de módulo aplicada sobre um valor

PUC-Rio - Certificação Digital Nº 1521382/CA

inteiro (comumente representada pelo operador % em linguagens de programação), ao passo que $(mod\ m)$ sucede uma congruência e indica sob qual módulo os dois valores precedentes são congruentes.

A aritmética modular possui inúmeras propriedades, entre elas o fato de que é possível realizar diversas operações matemáticas sobre dois valores inteiros sem que seja perdida a congruência entre eles. Para o RSA, são relevantes principalmente as seguintes propriedades, que serão utilizadas extensivamente na prova de corretude do algoritmo:

- Se $a \equiv b\ (mod\ m)$, então $a \cdot k \equiv b \cdot k\ (mod\ m)$ para qualquer número inteiro k.
- Se $a \equiv b\ (mod\ m)$, então $a^k \equiv b^k\ (mod\ m)$ para qualquer expoente natural k.

3.2.2 Utilização de Aritmética Modular pelo RSA

O RSA utiliza a operação de divisão modular nas suas funções de codificação e decodificação. Mais precisamente, estas funções realizam uma elevação a um expoente inteiro, seguida de divisão modular. Para este fim, o dado x e a cifra y são tratados também como números inteiros. Observe como são executadas as funções:

Codificação: $f(x) = x^e\ mod\ n$
Decodificação: $f^{-1}(y) = y^d\ mod\ n$

Já estabelecemos que o n é o produto de dois primos grandes p e q. Os expoentes e e d são determinados a partir de p e q pelo usuário que gera as chaves, e são denominados respectivamente expoente de codificação e expoente de decodificação. Os pares (n, e) e (n, d) são, respectivamente, as chaves pública e privada do algoritmo.

3.2.3 A Função Totiente de Euler

Para que o RSA funcione, é necessário que o expoente d seja capaz de decodificar as mensagens codificadas com o expoente e. Isto ocorre quando $d \cdot e \equiv 1\ \big(mod\ \phi(n)\big)$, sendo $\phi(n)$ a função totiente de Euler sobre n. (A explicação de por que exatamente esta congruência implica no funcionamento correto do algoritmo é bastante longa e encontra-se na descrição da prova de corretude do RSA, apresentada na seção 5.2.1. Por enquanto, a aceitaremos como uma convenção.)

A função $\phi(n)$ representa o número de inteiros entre 1 e n que são relativamente primos a n, isto é, não possuem nenhum divisor maior do que 1 em

PUC-Rio - Certificação Digital Nº 1521382/CA

comum com n. O Teorema de Euler afirma que, se a e b forem relativamente primos, então $a^{\phi(b)} \equiv 1\ (mod\ b)$. Utilizaremos esta equação para realizar a prova de corretude do RSA na seção 5.2.

3.2.4 Geração de Parâmetros e Codificação de Mensagens

Para gerar um par de chaves para RSA, um usuário deve seguir os seguintes passos:

- Escolher dois primos grandes p e q.
- Calcular $n = p \cdot q$.
- Calcular a função totiente $\phi(n) = (p-1) \cdot (q-1)$.
- Escolher um expoente e tal que $mdc\big(e, \phi(n)\big) = 1$. Esta propriedade garante que exista um d tal que $d \cdot e \equiv 1\ \big(mod\ \phi(n)\big)$.
- Calcular d.

Já estabelecemos que, dos parâmetros acima, são divulgados para os demais usuários apenas n e e, que, juntos, constituem a chave pública. Portanto, um adversário que porventura intercepte uma mensagem transmitida poderá ter acesso apenas a estes parâmetros, bem como à cifra y. Nota-se que o expoente de decodificação d pode ser calculado apenas caso se conheçam os valores de p e q. Assim sendo, a inviabilidade computacional do problema de fatoração de primos protege a chave privada e impede que ela seja calculada a partir dos parâmetros da chave pública. Ao mesmo tempo, d possui uma relação intrínseca com e que garante que o algoritmo funcione.

Para finalizar esta explicação, providenciamos um exemplo de geração de chaves, codificação, e decodificação em RSA. Este exemplo utiliza números primos pequenos para fins de simplicidade ilustrativa; reiteramos, no entanto, que uma execução real de RSA trabalha com primos da ordem de, pelo menos, 2^{512}.

Geração de chaves:

- Sejam $p = 5$ e $q = 13$
- $n = 5 \cdot 13 = 65$
- $\phi(n) = (5-1) \cdot (13-1) = 4 \cdot 12 = 48$
- $48 = 2^4 \cdot 3^1$; portanto, escolhemos $e = 5$ para satisfazer $mdc\big(e, \phi(n)\big) = 1$
- $d = 29$, já que $5 \cdot 29 = 145 \equiv 1\ (mod\ 48)$

Tendo obtido a chave pública $(65, 5)$, podemos realizar a operação de codificação:

- Seja $x = 32$
- $y = x^e\ mod\ n = 32^5\ mod\ 65 = 33554432\ mod\ 65 = 2$

PUC-Rio - Certificação Digital Nº 1521382/CA

Para recuperar x, basta realizar a decodificação com a chave privada $(65, 29)$:

- $x = y^d \; mod \; n = 2^{29} \; mod \; 65 = 536870912 \; mod \; 65 = 32$

A prova de funcionamento do RSA no Lean Theorem Prover é descrita na seção 5.2.

3.3 Blowfish

O algoritmo Blowfish é um algoritmo de criptografia com chave de bloco desenvolvido por Bruce Schneier. Assim como o DES, este algoritmo realiza 16 iterações de um mesmo passo de codificação sobre blocos de dados de 64 bits, operando simultaneamente sobre as metades esquerda e direita deste bloco, denominadas L e R respectivamente, cada uma com 32 bits. Ao contrário do DES, o Blowfish codifica todos os 64 bits de um bloco de dados em cada passo de codificação.

O algoritmo Blowfish utiliza 18 chaves P_1, P_2... P_{18}.

A codificação de um bloco de dados pelo Blowfish dá-se pela seguinte fórmula:

$$\left.\begin{aligned} L_{i+1} &= R_i \oplus F(L_i \oplus P_{i+1}) \\ R_{i+1} &= L_i \oplus P_{i+1} \end{aligned}\right\} \; 0 \leq i \leq 14$$

$$L_{16} = \left(R_{15} \oplus F(L_{15} \oplus P_{16})\right) \oplus P_{17}$$
$$R_{16} = (L_{15} \oplus P_{16}) \oplus P_{18}$$

A figura 3.5 ilustra o algoritmo Blowfish.

PUC-Rio - Certificação Digital Nº 1521382/CA

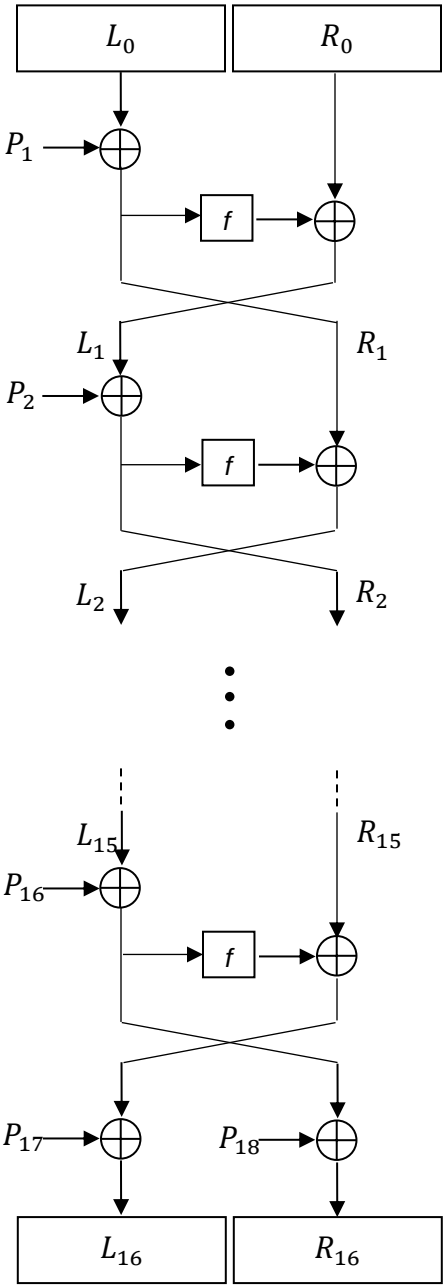

Fig 3.5: Estrutura do algoritmo Blowfish

A prova de funcionamento do algoritmo Blowfish no Lean Theorem Prover é descrita na seção 5.3.

PUC-Rio - Certificação Digital Nº 1521382/CA

4 Prova Interativa de Teoremas

Um assistente de provas interativo é, essencialmente, um ambiente de desenvolvimento que permite a um usuário descrever uma prova lógica ou matemática através de objetos e manipular estes objetos por meio de uma biblioteca pré-definida, utilizando assim provas já existentes para definição de outras mais complexas. Esta descrição de provas pode ser feita de diversos modos, seja através de algo semelhante a uma linguagem de programação, como é o caso do Lean Theorem Prover, seja por meio de uma interface gráfica.

A interatividade de um ambiente deste tipo manifesta-se no fato de o usuário não necessitar ter uma noção detalhada de como irá realizar cada passo da prova em questão antes de iniciar seu trabalho. Ele pode partir de uma ideia relativamente vaga, definindo no assistente de provas as hipóteses que deverá utilizar como ponto de partida bem como a conjectura que deseja provar a partir destas. A partir destas definições, o assistente é capaz de indicar a um usuário qual caminho ele deve seguir para alcançar uma prova da conjectura desejada. Veremos alguns exemplos de como isso pode ser realizado a seguir.

4.1 O Lean Theorem Prover

O Lean Theorem Prover, ou simplesmente Lean, é um assistente de provas interativo desenvolvido por Leonardo de Moura em 2012 e o framework utilizado para as formalizações de provas de teoremas apresentadas neste trabalho.

Como já mencionado, um assistente de provas interativo é um ambiente que permite a um usuário descrever a prova de um teorema através de uma interface. No caso do Lean, a descrição de provas e teoremas é feita por meio de código compilável, semelhante a uma linguagem de programação. Uma prova escrita nesta linguagem pode ser compilada e testada pelo programa, que informa ao usuário quaisquer erros ou inconsistências no código. Mais importante do que isto, porém, é que o Lean providencia uma representação de proposições lógicas (e dos teoremas compostos por elas) que permite ao usuário verificar facilmente por que uma determinada prova foi (ou não) realizada de maneira correta.

A representação de proposições lógicas é feita por meio de tipos ou variáveis. A seguir, um exemplo de definição de tipo e variável no Lean:

```
variable A : Type
variable a : A
```

O operador **:** indica que o objeto que está sendo declarado (aquele à sua esquerda) pertence a uma classe mais abrangente (aquela à direita do operador). A

PUC-Rio - Certificação Digital Nº 1521382/CA

classe `Type` é a mais abrangente na biblioteca do Lean. Estamos definindo uma variável `A` do tipo `Type` e uma variável `a` do tipo `A`.

Para se declarar múltiplas variáveis de uma só vez, utiliza-se o operador `variables`, no plural:

```
variables a b c : A
```

Uma vez definidas as variáveis, podemos definir funções que podem ser aplicadas sobre elas:

```
variable f (x : A) : Prop
variable g (x : A) : Prop
```

O que está sendo definido acima é que `f` e `g` são funções que têm como entrada objetos do tipo `A` e como saída objetos do tipo `Prop`, ou proposição lógica, este último já fazendo parte da biblioteca pré-definida do Lean. Podemos definir funções que retornem objetos de outros tipos, por exemplo:

```
variables A B : Type
variable a : A

variable f (x : A) : A
variable g (x : A) : B
```

Neste caso, estamos definindo dois tipos, `A` e `B`, e duas funções `f` e `g` que podem ser aplicadas sobre eles. Aqui, `f` é um objeto do tipo `A → A` (isto é, uma função que recebe um objeto do tipo `A` como entrada e retorna outro objeto do tipo `A` como saída) e `g` é um objeto do tipo `A → B`. As funções aplicadas sobre variáveis também são objetos; por exemplo, `f a` é um objeto do tipo `A` (pois considera-se que a variável de entrada da função é "cancelada", restando, portanto, apenas o tipo da variável de saída) e `g a` é um objeto do tipo `B`.

Tendo definido as variáveis que utilizaremos para compor provas de teoremas, resta definir nossas premissas – isto é, aquilo que tomamos como verdadeiro, por já ter sido comprovado (seja esta comprovação realizada dentro ou fora do contexto do nosso trabalho) e utilizamos como base para construir nossas provas.

Premissas são definidas no Lean da seguinte forma:

```
premise p1 : a = b
premise p2 : b = c
```

O operador de igualdade acima já existe incorporado na biblioteca do Lean e possui certas propriedades que serão descritas em breve. Basicamente, o que estamos

PUC-Rio - Certificação Digital Nº 1521382/CA

dizendo aqui é que existe um objeto `p1` do tipo `a = b` e um objeto `p2` do tipo `b = c`. Para o Lean, a existência de pelo menos um objeto de um determinado tipo de premissa equivale a dizer que a premissa é verdadeira. Neste caso, estamos dizendo que já conhecemos provas das proposições $a = b$ e $b = c$.

Agora, tendo definido nossas variáveis, funções e premissas, podemos enfim partir para a construção de teoremas no Lean. Isto se faz pelo meio do operador `theorem`. Após este operador, atribui-se um nome ao teorema, de forma semelhante à definição de variáveis e premissas. Em seguida, após o operador `:`, coloca-se, como tipo, a proposição que pretendemos provar, e, após o operador `:=`, o corpo da prova propriamente dita – isto é, como foi feita a prova.

Observe como se realiza a aplicação das propriedades de simetria e transitividade para construção de provas de $b = a$ e $a = c$ através daquelas que acabamos de definir. Reflexividade, simetria e transitividade existem na biblioteca do Lean, podendo ser invocadas por meio dos operadores `eq.refl`, `eq.symm` e `eq.trans`.

```
theorem t1 : b = a :=
eq.symm p1

theorem t2 : a = c :=
eq.trans p1 p2
```

Resta apenas esclarecer como o Lean diferencia provas realizadas corretamente daquelas que contêm erros. Se tentarmos realizar uma construção em que o tipo das variáveis e premissas não corresponde ao esperado, o Lean acusará o erro. O teorema abaixo, por exemplo, resultará em erro por tentar compor uma prova de $c = a$ aplicando a transitividade sobre as equações $a = b$ e $b = c$.

```
theorem t3 : c = a :=
eq.trans p1 p2
```

Isto conclui esta breve introdução ao funcionamento do Lean Theorem Prover. Mais comandos serão vistos adiante conforme se tornem relevantes.

PUC-Rio - Certificação Digital Nº 1521382/CA

5 Provas de Corretude e Consistência de Algoritmos de Criptografia no Lean Theorem Prover

Tendo apresentado os algoritmos e processos cujas provas constituem a base deste trabalho, bem como o framework a ser utilizado para verificação destas provas, podemos partir agora para a verificação propriamente dita.

As provas aqui descritas incluem todo o código desenvolvido no Lean Theorem Prover, acompanhado de comentários para facilitar o entendimento.

5.1 Provas de Corretude do *Data Encryption Standard*

Esta seção tem como objetivo descrever as provas de que os algoritmos do *Data Encryption Standard* descritos na seção 3.1 funcionam corretamente, e, subsequentemente, apresentar a forma como foi realizada a verificação destas provas no Lean.

A primeira coisa que devemos definir antes de descrever estas provas é a estrutura utilizada para representação dos dados na linguagem do Lean. O DES trabalha com blocos de dados de 32, 48, 56 ou 64 bits, dependendo do estágio do algoritmo. Para a representação destes blocos de dados como estruturas, definimos primeiro um tipo indutivo `bit`, composto por uma série de subtipos b_i definidos indutivamente, conforme demostrado abaixo:

```
inductive bit : Type :=
| b1 : bit
| b2 : bit
| b3 : bit
| b4 : bit
| b5 : bit
| b6 : bit
| b7 : bit
| b8 : bit
...
```

Por questões de espaço, o trecho de código transcrito acima compreende apenas uma parte do código produzido para a prova de corretude, que define subtipos de b_1 até b_{64} de forma a cobrir o número máximo de bits possíveis num bloco de dado no DES.

O que estes tipos representam é *um valor arbitrário, podendo ser 0 ou 1, que representa o valor do bit b_i em uma sequência de dados*. Estes valores são genéricos, e o fato de termos definido 64 tipos distintos para bits se deve apenas ao fato de estarmos trabalhando com sequências de no máximo 64 bits, podendo estes mesmos tipos serem utilizados em estruturas de dados que representem sequências menores. A ideia é apenas que, dados uma sequência s de k bits e inteiros i, j tal que $1 \leq i, j \leq k$,

PUC-Rio - Certificação Digital Nº 1521382/CA

o valor representado pelo tipo b_i seja *independente* daquele representado por b_j para todo $i \neq j$.

O que queremos demonstrar é que os valores dos bits antes da codificação de dados e os valores recuperados após a decodificação são iguais. Realizamos isto através de definição de tipos de modo a providenciar uma prova que seja válida para quaisquer possíveis valores de uma sequência de dados.

Agora que definimos um máximo de 64 valores independentes para bits, podemos utilizar o operador `structure` para definir estruturas de dados compostas por estes bits, como no exemplo a seguir:

```
structure data8 :=
(pos1 : bit) (pos2 : bit) (pos3 : bit) (pos4 : bit)
(pos5 : bit) (pos6 : bit) (pos7 : bit) (pos8 : bit)
```

O que estamos definindo acima é uma estrutura de dados representando sequências de 8 bits – algo semelhante ao conceito de classe em programação orientada a objetos, no sentido de que podemos definir instâncias desta estrutura que correspondam a sequências individuais de 8 bits. (Embora não utilizemos sequências tão curtas quando trabalhamos com algoritmos relacionados ao DES, esta definição é providenciada aqui apenas de modo a auxiliar com a explicação da prova).

Neste contexto de classes e objetos, os componentes $pos_1 \ldots pos_8$ podem ser comparados a funções associadas à classe `data8` que retornam o valor do bit situado na posição correspondente da sequência de dados. Como estamos trabalhando com tipos, este "valor" é, na realidade, um dos 64 subtipos de `bit` que acabamos de definir, sendo que a definição da instância especifica qual tipo está em cada posição. Para melhor ilustrar isto, vamos definir duas instâncias para a estrutura de dados *data8* acima:

```
definition d1 := data8.mk
bit.b1 bit.b2 bit.b3 bit.b4 bit.b5 bit.b6 bit.b7 bit.b8

definition d2 := data8.mk
bit.b9 bit.b10 bit.b11 bit.b12 bit.b13 bit.b14 bit.b15 bit.b16
```

Esta definição especifica que o objeto `d1` possui em sua primeira posição – aquela associada a `pos1` – um bit do tipo `b1`, seguido por um do tipo `b2` na segunda posição, e assim por diante. Para o objeto `d2`, temos uma definição análoga. Portanto, o comando `data8.pos1 d1` (aplicação da função `pos1` da "classe" `data8`, que retorna o valor da primeira posição de um objeto, sobre `d1`) deve ter como valor de retorno o tipo `bit.b1`.

A utilização dos tipos $b_1, b_2 \ldots b_{16}$ em ordem crescente é apenas por conveniência, já que, como explicado anteriormente, podemos instanciar uma classe usando quaisquer dentre os 64 tipos $b_1 \ldots b_{64}$ já definidos em qualquer ordem, sendo

PUC-Rio - Certificação Digital Nº 1521382/CA

importante apenas não utilizar o mesmo tipo duas vezes de modo a preservar a independência entre os bits. Como se pode ver, as duas sequências definidas acima possuem valores inteiramente independentes entre si.

Vejamos um exemplo de como se realizar uma prova de funcionamento de um algoritmo sobre uma dessas sequências de bits. Para isto, vamos primeiro definir uma estrutura mais curta, de 4 bits:

```
structure data4 :=
(pos1 : bit) (pos2 : bit) (pos3 : bit) (pos4 : bit)
```

Agora, vamos definir uma função que inverte as posições dos 4 bits de uma tal estrutura, de modo que o primeiro bit é trocado de posição com o quarto e o segundo é trocado de posição com o terceiro:

```
definition Reverse (d : data4) : data4 := data4.mk
(data4.pos4 d) (data4.pos3 d) (data4.pos2 d) (data4.pos1 d)
```

Como se pode ver, esta função recebe um objeto `d` do tipo `data4` e retorna um outro objeto `data4` com os 4 bits de `d` invertidos. É fácil ver que, ao aplicar esta função duas vezes sucessivamente a um mesmo objeto `d`, devemos ter como resultado a sequencia de bits inicial (isto é, `Reverse (Reverse d) = d`). Vamos provar isto no Lean, através do operador `theorem` utilizado para descrever a prova de um teorema.

```
theorem Double_Rev : d = Reverse (Reverse d) :=
eq.refl d
```

A operação `eq.refl d` utilizada acima é a operação de reflexividade da biblioteca do Lean. Trata-se de uma operação bastante útil: caso a prova que desejamos obter já esteja descrita em detalhe suficiente após o operador `:`, podemos utilizar este operador para comprovar a veracidade do teorema sem precisar descrever a prova em maior detalhe, pois o Lean é capaz de inferir como a prova deve ser realizada apenas por sua definição. Para provas mais complexas, seria necessário informar ao Lean como obtê-las, inserindo esta descrição após o operador `:=`. Isto será visto mais adiante.

Agora, se tentarmos realizar a seguinte prova:

```
theorem Single_Rev : d = Reverse d :=
eq.refl d
```

...o Lean acusará um erro, pois este “teorema” é falso. Afinal, a operação de inversão foi aqui aplicada apenas uma vez, e existem sequências de 4 bits – por exemplo, 0010 – que não são iguais quando lidas de trás para a frente. É assim que

PUC-Rio - Certificação Digital Nº 1521382/CA

utilizamos o Lean para descrever uma prova: caso consigamos fazê-lo sem que o provador acuse erro, podemos ter certeza que o teorema que descrevemos é verdadeiro.

5.1.1 Permutações Inicial e Final

Agora que já sabemos como realizar a prova de um teorema no Lean, podemos realizar nossa primeira prova relativa ao *Data Encryption Standard*: a prova de que a operação de permutação final, mostrada na tabela 3.2, "desfaz" a operação de permutação inicial, mostrada na tabela 3.1. Como ambas estas operações são simplesmente operações de reposicionamento de bits, isto é apenas uma versão mais complexa do problema descrito na seção 5.1.

Para realizar esta prova, primeiro devemos definir uma estrutura de dados representando uma sequência de 64 bits.

```
definition d := data64.mk
bit.b1  bit.b2  bit.b3  bit.b4  bit.b5  bit.b6  bit.b7  bit.b8
bit.b9  bit.b10 bit.b11 bit.b12 bit.b13 bit.b14 bit.b15 bit.b16
bit.b17 bit.b18 bit.b19 bit.b20 bit.b21 bit.b22 bit.b23 bit.b24
bit.b25 bit.b26 bit.b27 bit.b28 bit.b29 bit.b30 bit.b31 bit.b32
bit.b33 bit.b34 bit.b35 bit.b36 bit.b37 bit.b38 bit.b39 bit.b40
bit.b41 bit.b42 bit.b43 bit.b44 bit.b45 bit.b46 bit.b47 bit.b48
bit.b49 bit.b50 bit.b51 bit.b52 bit.b53 bit.b54 bit.b55 bit.b56
bit.b57 bit.b58 bit.b59 bit.b60 bit.b61 bit.b62 bit.b63 bit.b64
```

Em seguida, devemos definir as funções que representam as operações de permutação inicial e final. Note que as operações de reposicionamento de bits descritas a seguir correspondem àquelas demonstradas nas tabelas 3.1 e 3.2.

```
definition DES_InitialPerm (d : data64) : data64 := data64.mk
(data64.pos58 d) (data64.pos50 d) (data64.pos42 d) (data64.pos34 d)
(data64.pos26 d) (data64.pos18 d) (data64.pos10 d) (data64.pos2  d)
(data64.pos60 d) (data64.pos52 d) (data64.pos44 d) (data64.pos36 d)
(data64.pos28 d) (data64.pos20 d) (data64.pos12 d) (data64.pos4  d)
(data64.pos62 d) (data64.pos54 d) (data64.pos46 d) (data64.pos38 d)
(data64.pos30 d) (data64.pos22 d) (data64.pos14 d) (data64.pos6  d)
(data64.pos64 d) (data64.pos56 d) (data64.pos48 d) (data64.pos40 d)
(data64.pos32 d) (data64.pos24 d) (data64.pos16 d) (data64.pos8  d)
(data64.pos57 d) (data64.pos49 d) (data64.pos41 d) (data64.pos33 d)
(data64.pos25 d) (data64.pos17 d) (data64.pos9  d) (data64.pos1  d)
(data64.pos59 d) (data64.pos51 d) (data64.pos43 d) (data64.pos35 d)
(data64.pos27 d) (data64.pos19 d) (data64.pos11 d) (data64.pos3  d)
(data64.pos61 d) (data64.pos53 d) (data64.pos45 d) (data64.pos37 d)
(data64.pos29 d) (data64.pos21 d) (data64.pos13 d) (data64.pos5  d)
(data64.pos63 d) (data64.pos55 d) (data64.pos47 d) (data64.pos39 d)
(data64.pos31 d) (data64.pos23 d) (data64.pos15 d) (data64.pos7  d)

definition DES_FinalPerm (d : data64) : data64 := data64.mk
(data64.pos40 d) (data64.pos8  d) (data64.pos48 d) (data64.pos16 d)
```

PUC-Rio - Certificação Digital Nº 1521382/CA

```
(data64.pos56 d) (data64.pos24 d) (data64.pos64 d) (data64.pos32 d)
(data64.pos39 d) (data64.pos7  d) (data64.pos47 d) (data64.pos15 d)
(data64.pos55 d) (data64.pos23 d) (data64.pos63 d) (data64.pos31 d)
(data64.pos38 d) (data64.pos6  d) (data64.pos46 d) (data64.pos14 d)
(data64.pos54 d) (data64.pos22 d) (data64.pos62 d) (data64.pos30 d)
(data64.pos37 d) (data64.pos5  d) (data64.pos45 d) (data64.pos13 d)
(data64.pos53 d) (data64.pos21 d) (data64.pos61 d) (data64.pos29 d)
(data64.pos36 d) (data64.pos4  d) (data64.pos44 d) (data64.pos12 d)
(data64.pos52 d) (data64.pos20 d) (data64.pos60 d) (data64.pos28 d)
(data64.pos35 d) (data64.pos3  d) (data64.pos43 d) (data64.pos11 d)
(data64.pos51 d) (data64.pos19 d) (data64.pos59 d) (data64.pos27 d)
(data64.pos34 d) (data64.pos2  d) (data64.pos42 d) (data64.pos10 d)
(data64.pos50 d) (data64.pos18 d) (data64.pos58 d) (data64.pos26 d)
(data64.pos33 d) (data64.pos1  d) (data64.pos41 d) (data64.pos9  d)
(data64.pos49 d) (data64.pos17 d) (data64.pos57 d) (data64.pos25 d)
```

Agora, podemos realizar a prova de que a permutação final desfaz a permutação inicial, da mesma forma que fizemos com a operação `Double_Rev` na seção 5.1:

```
theorem Permutation_Rev : d = (DES_FinalPerm (DES_InitialPerm d)) :=
eq.refl d
```

Ao inserirmos este código no Lean e rodarmos o provador, não há acusação de erro. Isto comprova que a prova é verdadeira e que a operação de permutação final inverte a operação de permutação inicial.

5.1.2 Passo de Codificação

Uma vez tendo provado que as operações de permutação inicial e final se cancelam mutuamente, passamos para a demonstração de corretude de uma etapa de codificação propriamente dita, isto é, a série de operações ilustrada anteriormente na figura 3.2.

PUC-Rio - Certificação Digital Nº 1521382/CA

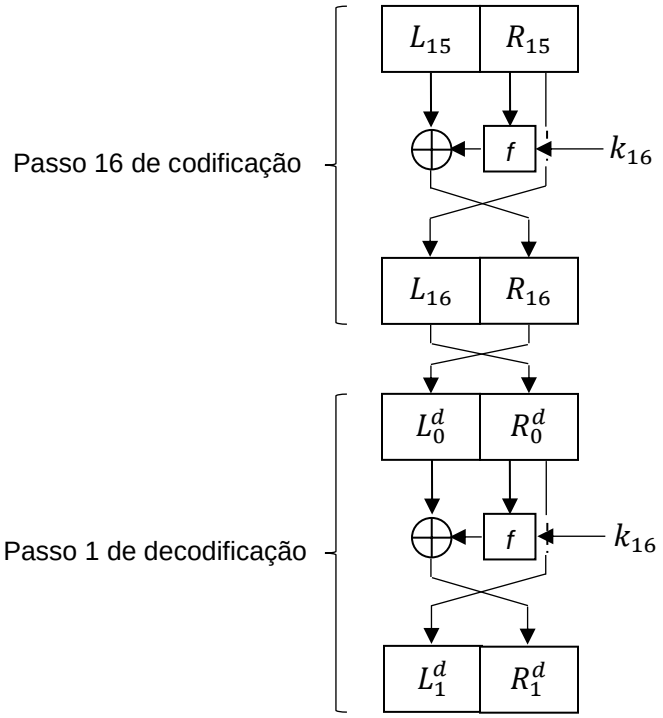

Fig 5.1: Último passo de uma sequência de codificação do DES, seguido do primeiro passo da sequência de decodificação

Queremos provar aqui que cada passo da sequência de codificação "desfaz" um passo da sequência de codificação. Como ilustrado na figura 5.1, utilizaremos para esta prova o primeiro passo da sequência de decodificação, por ser ele aplicado diretamente sobre o dado cifrado y. Demonstraremos que este passo é a operação inversa do 16º passo de codificação e recupera os dados L_{15} e R_{15} anteriores a este. Mais tarde, utilizaremos esta prova como caso base para a demonstração de corretude de uma sequência de codificação completa, através de indução. (Esta prova é descrita na seção 5.3).

Há alguns fatos para os quais é importante chamar a atenção antes de realizarmos a prova. Primeiramente, nota-se que ocorre uma inversão das posições de L_{16} e R_{16} após o 16º passo de codificação. Esta transposição é parte do processo de codificação do DES e sempre é realizada imediatamente antes da permutação final, tanto na codificação quanto na decodificação. Devido a isto, cada bloco de 64 bits codificado pelo DES encontra-se com suas metades invertidas em relação ao dado correspondente durante a codificação. Portanto, as igualdades que queremos demonstrar são, na realidade, $L_1^d = R_{15}$ e $R_1^d = L_{15}$. Na seção 5.3, é demonstrado que esta transposição é desfeita ao fim do processo de decodificação, retornando o dado ao seu estado original.

Em segundo lugar, nota-se que as operações de permutação inicial e final encontram-se omitidas tanto na figura 5.1 quanto na prova que segue. Esta omissão é proposital, devido ao fato de as operações de permutação serem, até certo ponto,

PUC-Rio - Certificação Digital Nº 1521382/CA

independentes da sequência de codificação, permitindo que sua prova de corretude seja realizada separadamente; de fato, demonstraremos que o DES funciona corretamente mesmo sem a aplicação das permutações.

Partamos então para a prova proposta. Conforme ilustrado na figura 5.1., temos como premissas que $L_{16}R_{16} = enc(L_{15}R_{15})$, $L_0^d R_0^d = R_{16}L_{16}$, e $L_1^d R_1^d = enc(L_0^d R_0^d)$, sendo enc um passo de codificação. Decompondo estas premissas, obtemos as seguintes hipóteses sobre as quais realizar a prova:

1. $L_{16} = R_{15}$
2. $R_{16} = L_{15} \oplus f(R_{15}, k_{16})$
3. $L_0^d = R_{16}$
4. $R_0^d = L_{16}$
5. $L_1^d = R_0^d$
6. $R_1^d = L_0^d \oplus f\left(R_0^d, k_{16}\right)$

O primeiro objetivo (isto é, a prova de $L_1^d = R_{15}$) é relativamente trivial. Aplicando sucessivamente a propriedade de transitividade às hipóteses 5, 4 e 1 acima, temos que

$$L_1^d = R_0^d = L_{16} = R_{15}$$

A prova de $R_1^d = L_{15}$ é um pouco mais complexa e necessita das propriedades de associatividade e idempotência da operação ou-exclusivo. É nesta prova que o fato de a mesma chave (k_{16}, neste exemplo) ser utilizada para codificação e decodificação se torna relevante, por permitir a eliminação de fatores através da idempotência.

$R_1^d = L_0^d \oplus f\left(R_0^d, k_{16}\right)$	(hipótese 6)
$= R_{16} \oplus f(L_{16}, k_{16})$	(substituição pelas hipóteses 3 e 4)
$= \left(L_{15} \oplus f(R_{15}, k_{16})\right) \oplus f(R_{15}, k_{16})$	(substituição pelas hipóteses 2 e 1)
$= L_{15} \oplus \left(f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16})\right)$	(associatividade de $\oplus$)
$= L_{15}$	(idempotência de $\oplus$)

Agora, veremos como são realizadas ambas estas provas no Lean Theorem Prover. A primeira coisa que devemos fazer é definir um tipo para representar dados, bem como as variáveis correspondentes aos blocos de dados, a chave k_{16}, e as funções *f* e ou-exclusivo. Os trechos de código a seguir mostram como essas definições são realizadas no Lean.

PUC-Rio - Certificação Digital Nº 1521382/CA

```
variable data : Type

variables
L15_E R15_E L16_E R16_E
L0_D  R0_D  L1_D  R1_D
K_16                    : data
```

Para definição das funções, colocamos os parâmetros de entrada entre parênteses após o nome da função:

```
variable xor (d1 d2 : data) : data
variable f (d k : data) : data
```

Em seguida, utilizamos o operador `premise` para definir as propriedades de associatividade e idempotência da operação ou-exclusivo:

```
premise xor_assoc (x1 x2 x3 : data) :
(xor (xor x1 x2) x3) = (xor x1 (xor x2 x3))

premise xor_cancel : ∀ x1 x2 : data, xor x1 (xor x2 x2) = x1
```

Agora, devemos definir no assistente de provas as seis hipóteses descritas anteriormente. Observe que todas possuem formato de igualdade. Lembramos que, no Lean, uma igualdade corresponde a um *tipo* e uma variável declarada para este tipo representa a existência de uma prova da igualdade em questão. Portanto, ao declarar uma variável `H1` do tipo `L16_E = R15_E`, por exemplo, estamos dizendo que conhecemos uma *prova* desta igualdade. Declaramos, então, uma variável para cada tipo corresponde a uma das seis igualdades que representam nossas premissas:

```
premise H1        : L16_E = R15_E
premise H2        : R16_E = xor L15_E (f R15_E K_16)
premise H3        : L0_D  = R16_E
premise H4        : R0_D  = L16_E
premise H5        : L1_D  = R0_D
premise H6        : R1_D  = xor L0_D (f R0_D K_16)
```

Tendo definido nossas variáveis e premissas, podemos agora partir para a prova propriamente dita. Primeiramente, realizemos a prova mais simples, de $L_1^d = R_{15}$. O que queremos demonstrar é que a existência das seis variáveis, ou hipóteses, acima implica na existência de uma variável do tipo `R15_E = L1_D`, correspondente à igualdade que queremos demonstrar.

Para obter a variável desejada, dispomos das funções `eq.symm` e `eq.trans` providenciadas pelo Lean. Estas funções representam, respectivamente, as propriedades de simetria e transitividade, e podem ser aplicadas a qualquer objeto igualdade. Por exemplo, caso possuamos uma premissa `H` do tipo `a = b`, a aplicação de `eq.symm` sobre `H` retorna uma prova de `b = a`. A funções de transitividade

PUC-Rio - Certificação Digital Nº 1521382/CA

`eq.trans`, por sua vez, recebe dois argumentos de tipos `a = b` e `b = c`, e retorna uma prova de `a = c`.

Abaixo, é demonstrado, passo-a-passo, como se obtém a prova de $L_1^d = R_{15}$, com comentários na coluna da direita para facilitar o acompanhamento. O comando *have* é utilizado para obtenção de provas temporárias – neste caso, `H7`, `H8`, `H9` e `H10` – que podem ser vistas como sub-objetivos definidos de forma a demarcar o caminho até a prova final.

Código	Comentários
`theorem ProofRight_Long : R15_E = L1_D :=`	
`have H7 : R15_E = L16_E, from eq.symm H1,`	Obtenção de prova de $R_{15} = L_{16}$ através da simetria de H1
`have H8 : L16_E = R0_D, from eq.symm H4,`	Prova de $L_{16} = R_0^d$ através da simetria de H4
`have H9 : R0_D = L1_D, from eq.symm H5,`	Prova de $R_0^d = L_1^d$ através da simetria de H5
`have H10 : R15_E = R0_D, from eq.trans H7 H8,`	Prova de $R_{15} = R_0^d$ através da transitividade sobre H7 e H8
`show R15_E = L1_D, from eq.trans H10 H9`	Prova de $R_{15} = L_1^d$ através da transitividade sobre H10 e H9

Tabela 5.1: Prova de corretude da metade esquerda de um passo de codificação do DES, ou $L_1^d = R_{15}$

Embora a definição de hipóteses preliminares dentro do teorema facilite o acompanhamento da prova passo-a-passo, ela não é estritamente necessária. Podemos, inclusive, realizar a prova em uma única linha, aplicando os comandos `eq.symm` e `eq.trans` sucessivamente sobre as hipóteses pré-definidas:

```
theorem ProofRight_Short : R15_E = L1_D :=
eq.trans (eq.trans (eq.symm H1) (eq.symm H4)) (eq.symm H5)
```

Também é possível utilizar a notação disponível na biblioteca `eq.ops` do Lean para produzir provas ainda mais compactas. O operador `⁻¹` desta biblioteca equivale a uma versão abreviada da chamada `eq.symm`, ao passo que o ponto `·` equivale à operação de transitividade `eq.trans`. Como se pode ver abaixo, a prova produzida é relativamente curta e de fácil leitura quando visualizada juntamente à definição das hipóteses `H1`, `H4` e `H5`.

```
open eq.ops
```

PUC-Rio - Certificação Digital Nº 1521382/CA

```
theorem ProofRight_Shorter : R15_E = L1_D :=
(H1⁻¹ · H4⁻¹) · H5⁻¹
```

Agora, vamos realizar a prova de $L_{15} = R_1^d$. Utilizamos aqui outro comando, `eq.subst`. Este comando permite realizar a substituição de uma variável ou equação a por outra equação b em uma equação que contenha a, desde que possuamos uma prova de $a = b$.

Código	Comentários
`theorem ProofLeft : L15_E = R1_D :=`	
`have H11 : R1_D = xor L0_D (f R0_D K_16), from H6,`	$R_1^d = L_0^d \oplus f(R_0^d, k_{16})$, de H6
`have H12 : R1_D = xor R16_E (f R0_D K_16), from eq.subst H3 H11,`	Prova de $R_1^d = R_{16} \oplus f(R_0^d, k_{16})$ através da substituição de H3 em H11
`have H13 : R1_D = xor R16_E (f L16_E K_16), from eq.subst H4 H12,`	Prova de $R_1^d = R_{16} \oplus f(L_{16}, k_{16})$ através da substituição de H4 em H12
`have H14 : R1_D = xor R16_E (f R15_E K_16), from eq.subst H1 H13,`	Prova de $R_1^d = R_{16} \oplus f(R_{15}, k_{16})$ através da substituição de H1 em H13
`have H15 : R1_D = xor (xor L15_E (f R15_E K_16)) (f R15_E K_16), from eq.subst H2 H14,`	Prova de $R_1^d = \left(L_{15} \oplus f(R_{15}, k_{16})\right) \oplus f(R_{15}, k_{16})$ através da substituição de H2 em H14
`have H16 : xor (xor L15_E (f R15_E K_16)) (f R15_E K_16) = xor L15_E (xor (f R15_E K_16) (f R15_E K_16)), from xor_assoc L15_E (f R15_E K_16) (f R15_E K_16),`	Prova de $\left(L_{15} \oplus f(R_{15}, k_{16})\right) \oplus f(R_{15}, k_{16}) = L_{15} \oplus \left(f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16})\right)$ através da associatividade de $\oplus$
`have H17 : R1_D = xor L15_E (xor (f R15_E K_16) (f R15_E K_16)), from eq.trans H15 H16,`	Prova de $R_1^d = L_{15} \oplus \left(f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16})\right)$ através da transitividade sobre H15 e H16
`have H18 : xor L15_E (xor (f R15_E K_16) (f R15_E K_16)) = L15_E, from xor_cancel L15_E (f R15_E K_16),`	Prova de $L_{15} \oplus \left(f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16})\right) = L_{15}$ através da idempotência de $\oplus$
`show L15_E = R1_D, from eq.symm (eq.trans H17 H18)`	Prova de $L_{15} = R_1^d$ através da transitividade sobre H17 e H18 e da simetria sobre o resultado

Tabela 5.2: Prova de corretude da metade direita de um passo de codificação do DES, ou $L_{15} = R_1^d$

PUC-Rio - Certificação Digital Nº 1521382/CA

5.1.3 Sequência de Codificação

Nas duas seções anteriores, foram demonstradas as seguintes propriedades do DES: primeiro, que as permutações inicial e final do DES se cancelam, e que a primeira etapa de decodificação recupera o valor do dado antes da 16ª etapa de codificação, embora com as posições de esquerda e direita invertidas. Vamos agora demonstrar, por indução, que os 16 passos da etapa de decodificação do DES invertem os 16 passos da etapa de codificação, ou seja, que $L_{16}^{d} = R_0$ e $R_{16}^{d} = L_0$.

Por conveniência, os tipos utilizados para representação de dados (L_i, R_i, L_i^d e R_i^d) são definidos como funções tendo i como valor de entrada. Isto é feito de modo a facilitar a aplicação do passo indutivo.

```
variable L  (n : nat) : data
variable R  (n : nat) : data
variable Ld (n : nat) : data
variable Rd (n : nat) : data
```

Em seguida, novamente definimos as premissas que usaremos como base para a prova.

```
premise BaseLeft  : Ld 0 = R 16
premise BaseRight : Rd 0 = L 16
premise StepLeft  :
∀ i : nat, Ld i = R (16 - i) → Ld (i + 1) = R (16 - (i + 1))
premise StepRight :
∀ i : nat, Rd i = L (16 - i) → Rd (i + 1) = L (16 - (i + 1))
```

A seguir, descreve-se a prova por indução no Lean Theorem Prover de que $L_{16}^{d} = R_0$.

PUC-Rio - Certificação Digital Nº 1521382/CA

Código	Comentários

`theorem ProofLeft : Ld 16 = R 0 :=`
 	
`have H0 : Ld 0 = R 16,`
`from BaseLeft,`
 	
`have H1 : Ld ( 0 + 1) = R (16 - ( 0 + 1)), from (StepLeft 0) H0,`
`have H2 : Ld ( 1 + 1) = R (16 - ( 1 + 1)), from (StepLeft 1) H1,`
`have H3 : Ld ( 2 + 1) = R (16 - ( 2 + 1)), from (StepLeft 2) H2,`
`have H4 : Ld ( 3 + 1) = R (16 - ( 3 + 1)), from (StepLeft 3) H3,`
`have H5 : Ld ( 4 + 1) = R (16 - ( 4 + 1)), from (StepLeft 4) H4,`
`have H6 : Ld ( 5 + 1) = R (16 - ( 5 + 1)), from (StepLeft 5) H5,`
`have H7 : Ld ( 6 + 1) = R (16 - ( 6 + 1)), from (StepLeft 6) H6,`
`have H8 : Ld ( 7 + 1) = R (16 - ( 7 + 1)), from (StepLeft 7) H7,`
`have H9 : Ld ( 8 + 1) = R (16 - ( 8 + 1)), from (StepLeft 8) H8,`
`have H10 : Ld ( 9 + 1) = R (16 - ( 9 + 1)), from (StepLeft 9) H9,`
`have H11 : Ld (10 + 1) = R (16 - (10 + 1)), from (StepLeft 10) H10,`
`have H12 : Ld (11 + 1) = R (16 - (11 + 1)), from (StepLeft 11) H11,`
`have H13 : Ld (12 + 1) = R (16 - (12 + 1)), from (StepLeft 12) H12,`
`have H14 : Ld (13 + 1) = R (16 - (13 + 1)), from (StepLeft 13) H13,`
`have H15 : Ld (14 + 1) = R (16 - (14 + 1)), from (StepLeft 14) H14,`
`have H16 : Ld (15 + 1) = R (16 - (15 + 1)), from (StepLeft 15) H15,`
 	
`show Ld 16 = R 0, from H16`
	Caso base
	
	Aplicação dos 16 passos indutivos
	
	Prova de $R_{16}^{d} = L_0$ por indução

Tabela 5.3: Prova de $L_{16}^{d} = R_0$ para o algoritmo DES

A prova de $R_{16}^{d} = L_0$é realizada de forma análoga, apenas substituindo as funções `Ld`, `R`, `BaseLeft` e `StepLeft` por seus equivalentes previamente definidos.

Temos agora provas de $L_{16}^{d} = R_0$ e $R_{16}^{d} = L_0$; por extensão, acabamos de demonstrar que $L_0R_0 = R_{16}^{d}L_{16}^{d}$.

PUC-Rio - Certificação Digital Nº 1521382/CA

Como as duas metades do bloco de dados do DES são trocadas de posição após o 16º passo da etapa de decodificação, recuperamos L_0R_0, a mensagem original. Isto comprova o funcionamento do DES.

5.2 Provas de Corretude do Algoritmo RSA

São apresentadas aqui a prova de corretude do algoritmo RSA descrito na seção 3.2. Assim como nas seções dedicadas ao DES, é apresentada primeiro a prova realizada no papel, e, em seguida, a tradução desta para a linguagem do Lean.

5.2.1 Prova

$y = enc(x) = x^e \ mod \ n$, por definição do dado cifrado y
$dec(y) = y^d \ mod \ n$

Para provar que o algoritmo funciona conforme esperado, basta demonstrar que o valor obtido após a operação de decodificação, $dec(y)$, é igual ao valor do dado original x para quaisquer valores de x, n, d e e. Portanto, temos como objetivo provar que:

$x = dec(y) = dec(x^e \ mod \ n) = (x^e \ mod \ n)^d \ mod \ n$

Como propriedade da operação de divisão modular, temos que $(a^b \ mod \ n)^c \ mod \ n = a^{bc} \ mod \ n$; o que reduz nosso objetivo a provar $x \equiv x^{de} \ (mod \ n)$. Esta prova pode ser dividida em dois casos, dependendo do máximo divisor comum entre x e n.

Primeiro caso: $mdc(x, n) = 1$

Por aplicação do teorema de Euler, temos que $x^{\phi(n)} \equiv 1 \ (mod \ n)$, de onde segue que

$\left(x^{\phi(n)}\right)^t \equiv 1^t \ (mod \ n)$ (elevação de ambos os lados a um expoente t)
$\left(x^{\phi(n)}\right)^t \equiv 1 \ (mod \ n)$
$x \cdot \left(x^{\phi(n)}\right)^t \equiv x \ (mod \ n)$ (multiplicação por x)
$x \cdot x^{t \cdot \phi(n)} \equiv x \ (mod \ n)$
$x^{1+t \cdot \phi(n)} \equiv x \ (mod \ n)$

Como d e e são gerados de forma a garantir que $d \cdot e = 1 + t \cdot \phi(n)$, acabamos de provar que $x \equiv x^{de} \ (mod \ n)$.

PUC-Rio - Certificação Digital Nº 1521382/CA

Segundo caso: $mdc(x, n) \neq 1$

Este caso é significativamente mais complexo do que o anterior. Primeiramente, deve-se observar que possuímos as seguintes premissas:

- n é o produto de dois primos p e q;
- x possui algum divisor maior do que 1 em comum com n.

Como a fatoração de n em primos consiste apenas de p e q, podemos ter certeza que x possui p ou q como fator. Além disso, como x é necessariamente menor do que n, podemos afirmar que x não pode possuir ambos os primos como fatores.

Como não definimos nenhuma propriedade que diferencie p de q, podemos escolher um destes como fator hipotético de x para realizar esta prova. Seja, então, por definição, $x = r \cdot p$ para algum r. Portanto, temos que $mdc(x, q) = 1$.

Vejamos então como é alcançada a prova de $x \cdot \left(x^{\phi(n)}\right)^t \equiv x \ (mod\ n)$:

$$mdc(x, q) = 1$$
$$x^{\phi(q)} \equiv 1 \ (mod\ q)$$
$$\left(x^{\phi(q)}\right)^t \equiv 1^t \ (mod\ q)$$
$$\left(x^{\phi(q)}\right)^t \equiv 1 \ (mod\ q)$$
$$\left(\left(x^{\phi(q)}\right)^t\right)^{(p-1)} \equiv 1^{(p-1)} \ (mod\ q)$$
$$\left(\left(x^{\phi(q)}\right)^t\right)^{(p-1)} \equiv 1 \ (mod\ q)$$
$$\left(\left(x^{\phi(q)}\right)^{(p-1)}\right)^t \equiv 1 \ (mod\ q)$$
$$\left(\left(x^{\phi(q)}\right)^{(p-1)}\right)^t \equiv 1 \ (mod\ q)$$
$$\left(x^{\phi(q)\cdot(p-1)}\right)^t \equiv 1 \ (mod\ q)$$
$$\left(x^{(q-1)\cdot(p-1)}\right)^t \equiv 1 \ (mod\ q)$$
$$\left(x^{\phi(n)}\right)^t \equiv 1 \ (mod\ q)$$
$$\left(x^{\phi(n)}\right)^t = 1 + (v \cdot q)$$
$$x \cdot \left(x^{\phi(n)}\right)^t = x + (x \cdot v \cdot q) = x + (w \cdot p \cdot v \cdot q) = x + (w \cdot v) \cdot n$$
$$x \cdot \left(x^{\phi(n)}\right)^t \equiv x \ (mod\ n)$$

Para o caso em que x possui q como fator ao invés de p, a prova é análoga, apenas trocando p por q.

Já vimos no primeiro caso como chegar a $x \equiv x^{de} \ (mod\ n)$ a partir de $x \cdot \left(x^{\phi(n)}\right)^t \equiv x \ (mod\ n)$, portanto, está concluída a prova para ambos os casos. Está demonstrado, então, que a operação de decodificação do RSA recupera o dado x existente antes da operação de codificação.

PUC-Rio - Certificação Digital Nº 1521382/CA

5.2.2 Prova no Lean Theorem Prover

Antes de ser realizada a transcrição da prova demonstrada na seção 5.2.1 para a linguagem do Lean, devemos definir no código do assistente de provas todas as variáveis, funções, premissas e propriedades que serão necessárias.

Em primeiro lugar, devemos declarar as variáveis. Observando as provas, vemos que são utilizadas, ao todo, seis variáveis, todas representando números naturais: x, n, p, q, d e e. Ao contrário do que foi realizado na prova para o DES descrita na seção 5.1.2, optamos aqui por definir estas variáveis como pertencendo ao tipo `nat` já existente na biblioteca do Lean. Esta escolha é feita apenas por conveniência, por permitir que utilizemos as propriedades de comutatividade e associatividade de multiplicação de naturais definidas na biblioteca; poderíamos também optar por definir um novo tipo e especificar estas propriedades externamente.

```
open nat

variables one x n p q d e : nat
```

Note que o número 1 foi definido acima como uma variável. Isto pode ser feito porque o número 1 tem importância aqui apenas no que diz respeito às propriedades que se aplicam a ele, que definiremos em breve.

Definidas as variáveis, o próximo passo é definir as funções sobre números naturais. Primeiro, definimos a função `gcd`, que representa o máximo divisor comum entre a e b, e a função `phi`, que representa a função totiente de Euler $\phi(n)$, correspondente ao número de inteiros entre 1 e n que não possuem divisores maiores do que 1 em comum com n.

```
variable gcd (a b : nat) : nat
variable phi (n   : nat) : nat
```

Em seguida, as propriedades `prime` e `congruent`, que retornam um valor do tipo `Prop`, podendo ser verdadeiro ou falso. Neste contexto, `prime a` significa que a é primo, e `congruent a b n` equivale à afirmação $a \equiv b \ (mod\ n)$. A existência de uma premissa `H : congruent a b n` equivale a uma prova de que esta congruência é verdadeira.

```
variable prime     (a           : nat) : Prop
variable congruent (a b modulus : nat) : Prop
```

Em seguida, definimos as premissas sobre as quais construiremos nossa prova. Na coluna direita do quadro abaixo, são providenciadas explicações de cada premissa para facilitar sua compreensão:

PUC-Rio - Certificação Digital Nº 1521382/CA

Código no Lean Theorem Prover	Premissa
`premise nDef : n = p * q`	$n = p \cdot q$, por definição
`premise pPhi : phi p = p - one`	$\phi(p) = p - 1$
`premise qPhi : phi q = q - one`	$\phi(q) = q - 1$
`premise nPhi : phi n = (q - one) * (p - one)`	$\phi(n) = (q-1) \cdot (p-1)$
`premise xLess : x < n`	$x < n$, por definição
`premise pIsPrime : prime p`	p é primo
`premise qIsPrime : prime q`	q é primo
`premise de_Inverse : (congruent (d * e) one (phi n))`	$d \cdot e \equiv 1 (mod\ \phi(n))$

Tabela 5.4: Definição no Lean de premissas sobre as variáveis `one`, `x`, `n`, `p`, `q`, `d`, e `e` para a prova de corretude do algoritmo RSA

Agora, definiremos algumas propriedades do número 1 que são relevantes para a nossa prova:

Código no Lean Theorem Prover	Propriedade
`premise OneMul : ∀ n : nat, one * n = n`	$\forall n, 1 \cdot n = n$
`premise OneExp : ∀ n : nat, one ^ n = one`	$\forall n, 1^n = 1$
`premise ExpOne : ∀ n : nat, n ^ one = n`	$\forall n, n^1 = n$

Tabela 5.5: Definição no Lean de propriedades do número 1 para a prova de corretude do algoritmo RSA

PUC-Rio - Certificação Digital Nº 1521382/CA

Algumas propriedades da exponenciação de inteiros:

Código no Lean Theorem Prover	Propriedade
`premise ExpSum (a b c : nat) : (a ^ b) * (a ^ c) = a ^ (b + c)`	$a^{\mathrm{b}} \cdot a^{c} = a^{\mathrm{b}+c}$
`premise ExpMul (a b c : nat) : (a ^ b) ^ c = a ^ (b * c)`	$\left(a^{\mathrm{b}}\right)^{c} = a^{\mathrm{b}\cdot c}$
`premise ExpSwap (a b c : nat) : (a ^ b) ^ c = (a ^ c) ^ b`	$\left(a^{\mathrm{b}}\right)^{c} = (a^{\mathrm{c}})^{b}$

Tabela 5.6: Definição no Lean de propriedades da exponenciação para a prova de corretude do algoritmo RSA

A definição do operador de módulo e algumas de suas propriedades:

Código no Lean Theorem Prover	Significado
`premise ModuloDef (a b n : nat) : congruent a b n ↔ ∃ c, a = b + (c * n)`	$a \equiv b\ (mod\ n)$ se e somente se $\exists \mathrm{c}, \mathrm{a} = \mathrm{b} + (c \cdot n)$
`premise CongruenceReflexivity (a b n : nat) : congruent a b n → congruent b a n`	Se $a \equiv b\ (mod\ n)$, Então $b \equiv a\ (mod\ n)$ (reflexividade)
`premise CongruenceScaling (a b n k : nat) : congruent a b n → congruent (a * k) (b * k) n`	Se $a \equiv b\ (mod\ n)$, então $(a \cdot k) \equiv (b \cdot k)\ (mod\ n)$ (escalabilidade por multiplicação)
`premise CongruenceExponentiation (a b n k : nat) : congruent a b n → congruent (a ^ k) (b ^ k) n`	Se $a \equiv b\ (mod\ n)$, então $a^{\mathrm{k}} \equiv b^{\mathrm{k}}\ (mod\ n)$ (exponenciação)

Tabela 5.7: Definição no Lean do operador de módulo e suas propriedades para a prova de corretude do algoritmo RSA

PUC-Rio - Certificação Digital Nº 1521382/CA

As propriedades de escalabilidade e distributividade da multiplicação:

Código no Lean Theorem Prover	Propriedade

`premise EqMul (a b k : nat) :`
`a = b → k * a = k * b`
 	
`premise MulDistrib (a b c : nat) :`
`a * (b + c) = (a * b) + (a * c)`
	Se $a = b$,
	então $k \cdot a = k \cdot b$
	
	$a \cdot (b + c) =$
	$(a \cdot b) + (a \cdot c)$

Tabela 5.8: Definição no Lean das propriedades de escalabilidade e distributividade da multiplicação para a prova de corretude do algoritmo RSA

Por fim, definimos o teorema de Euler e uma propriedade do máximo divisor comum que será necessária:

Código no Lean Theorem Prover	Propriedade

`premise Euler (a b : nat) :`
`gcd a b = one → congruent (a ^ (phi b)) one b`
 	
`premise GCDProperty (a b c : nat) :`
`gcd a (b * c) ≠ one → prime b → prime c → a < b * c →`
`(((∃ n, a = n * b) ∧ (gcd a c = one)) ∨`
`((∃ n, a = n * c) ∧ (gcd a b = one)))`
	Se $mdc(a, b) = 1$,
	então $a^{\phi(b)} \equiv 1\ (mod\ b)$
	
	Se b e c são primos,
	$a > b \cdot c$, e a possui um divisor maior do que 1 em comum com $b \cdot c$,
	então existe um n tal que $a = n \cdot b$ e $mdc(a, c) = 1$, ou existe um n tal que $a = n \cdot c$ e $mdc(a, b) = 1$

Tabela 5.9: Definição no Lean do teorema de Euler e de uma propriedade do máximo divisor comum necessária para a prova de corretude do algoritmo RSA

Finalmente, terminamos de definir as variáveis, funções e premissas que serão necessárias para realizar a prova de funcionamento do RSA. Vejamos agora, nas tabelas a seguir como foi feita a prova propriamente dita.

A primeira tabela, vista a seguir, cobre o primeiro caso descrito na seção 5.2.1, no qual $mdc(x, n) = 1$, ou, em outras palavras, o dado x e o módulo de divisão n são primos entre si.

PUC-Rio - Certificação Digital Nº 1521382/CA

Código	Comentários
`theorem ProofCoprime (t : nat) : gcd x n = one → congruent (x * ((x ^ (phi n)) ^ t)) x n :=`	
`assume H1 : gcd x n = one,`	Hipótese inicial, i.e., $mdc(x,n) = 1$
`have H2 : congruent (x ^ (phi n)) one n,` `from Euler x n H1,`	Prova de $x^{\phi(n)} \equiv 1\ (mod\ n)$ pela aplicação do teorema de Euler sobre H1

`have Hexpt : congruent (x ^ (phi n)) one n →`
`congruent ((x ^ (phi n)) ^ t) (one ^ t) n,`
`from CongruenceExponentiation (x ^ (phi n)) one n t,`
	Definição de uma hipótese preliminar, $x^{\phi(n)} \equiv 1\ (mod\ n) \rightarrow \left(x^{\phi(n)}\right)^t \equiv 1^t\ (mod\ n)$, através da propriedade de conservação de congruência após exponenciação

`have H3 : congruent ((x ^ (phi n)) ^ t) (one ^ t) n,` `from Hexpt H2,`	Obtenção da prova de $\left(x^{\phi(n)}\right)^t \equiv 1^t\ (mod\ n)$ através da aplicação de Hexpt sobre H2
`have Honet : one ^ t = one,` `from OneExp t,`	Definição de hipótese preliminar $1^t \equiv 1$ por propriedade de potências com base 1
`have H4 : congruent ((x ^ (phi n)) ^ t) one n,` `from eq.subst Honet H3,`	Prova de $\left(x^{\phi(n)}\right)^t \equiv 1\ (mod\ n)$ por substituição de $1^t \equiv 1$ em H3

`have Hmulx : congruent ((x ^ (phi n)) ^ t) one n →`
`congruent (((x ^ (phi n)) ^ t) * x) (one * x) n,`
`from CongruenceScaling ((x ^ (phi n)) ^ t) one n x,`
	Hipótese preliminar $\left(x^{\phi(n)}\right)^t \equiv 1\ (mod\ n) \rightarrow \left(x^{\phi(n)}\right)^t \cdot x \equiv 1 \cdot x\ (mod\ n)$ através da propriedade de conservação de congruência após multiplicação

`have H5 : congruent (((x ^ (phi n)) ^ t) * x) (one * x) n,` `from Hmulx H4,`	Prova de $\left(x^{\phi(n)}\right)^t \cdot x \equiv 1 \cdot x\ (mod\ n)$ por aplicação de Hmulx sobre H4
`have Honex : one * x = x,` `from OneMul x,`	Obtenção de hipótese $1 \cdot x = x$ pela propriedade de multiplicação por 1

PUC-Rio - Certificação Digital Nº 1521382/CA

`have H6 : congruent (((x ^ (phi n)) ^ t) * x) x n,`
`  from eq.subst Honex H5,`
 	
`show congruent (x * ((x ^ (phi n)) ^ t)) x n,`
`  from eq.subst (mul.comm ((x ^ (phi n)) ^ t) x) H6`
	Substituição de $1 \cdot x = x$ em H5; obtenção de prova de $\left(x^{\phi(n)}\right)^t \cdot x \equiv x \ (mod\ n)$
	
	Prova de $x \cdot \left(x^{\phi(n)}\right)^t \equiv x \ (mod\ n)$ por comutatividade da multiplicação

Tabela 5.10: Prova de corretude do funcionamento do algoritmo RSA no Lean Theorem Prover, para o caso em que $mdc(x, n) = 1$

Nota-se que a prova formalizada acima utiliza extensivamente definição de hipóteses preliminares da forma $a \rightarrow b$ ou $a = b$, como a hipótese `Hexpt`. Estas hipóteses distinguem-se daquelas que representam os passos do teorema principal por não possuírem rótulos numerados. A definição destas hipóteses temporárias é opcional, podendo elas serem também definidas na mesma linha em que é realizada a sua aplicação sobre os passos principais, sem necessidade de atribuição de um rótulo. (Isto será explorado em maior detalhe mais adiante). O teorema acima foi redigido desta forma de modo a diminuir a complexidade de cada passo da prova, evitando o acúmulo de parâmetros após os comandos `from`. No entanto, a opção por esta notação aumenta significativa o tamanho da prova, o que pode ser um problema para teoremas mais extensos.

Para a maior parte da prova do caso em que $mdc(x, n) \neq 1$, foi utilizada uma notação mais concisa, o que será explicado em breve. Primeiramente, porém, devemos mencionar que a prova deste segundo caso se encontra dividida em duas partes devido ao seu tamanho. O bloco de código a seguir representa apenas a primeira parte, que descreve como se obtém, a partir da hipótese $mdc(x, n) \neq 1$, uma prova da fórmula lógica

$$\left((\exists n, x = n * p) \wedge (mdc(x, q) = 1)\right) \vee \left((\exists n, x = n * q) \wedge (mdc(x, p) = 1)\right)$$

Esta fórmula é a representação lógica das proposições descritas no segundo caso da seção 5.2.1. Ela afirma que, dentre os dois números primos p e q, o dado de entrada x é múltiplo de um e apenas um destes, sendo relativamente primo ao outro (isto é, possuindo mdc igual a 1). Vejamos como a obtenção desta fórmula a partir da hipótese é feita no Lean:

PUC-Rio - Certificação Digital Nº 1521382/CA

Código	Comentários

`theorem ProofNotCoprime_Part1 :`
`gcd x n ≠ one →`
`(((∃ n, x = n * p) ∧ (gcd x q = one)) ∨`
`((∃ n, x = n * q) ∧ (gcd x p = one))) :=`	

`assume H1 : gcd x n ≠ one,`	Hipótese inicial, i.e., $mdc(x,n) \neq 1$
`have H2 : x < p * q,` `from eq.subst nDef xLess,`	
`have H3 : gcd x (p * q) ≠ one,` `from eq.subst nDef H1,`	Prova de $mdc(x, p*q) \neq 1$ pela definição de n

`show (((∃ n, x = n * p) ∧ (gcd x q = one)) ∨`
`((∃ n, x = n * q) ∧ (gcd x p = one))),`
`from GCDProperty x p q H3 pIsPrime qIsPrime H2`
	Prova de $\left(\begin{matrix}(\exists n, x = n*p)\\ \wedge\,(mdc(x,q)=1)\end{matrix}\right) \vee \left(\begin{matrix}(\exists n, x = n*q)\\ \wedge\,(mdc(x,p)=1)\end{matrix}\right)$ pelas propriedades definidas anteriormente

Tabela 5.11: Primeira parte da prova de corretude do funcionamento do algoritmo RSA no Lean Theorem Prover, para o caso em que $mdc(x,n) \neq 1$

Prosseguimos agora para a segunda parte do teorema, que visa obter, a partir da fórmula demonstrada acima, uma prova de $x \cdot \left(x^{\phi(n)}\right)^t \equiv x\ (mod\ n)$. Para começar, Escolhemos a metade esquerda da fórmula, ou $(\exists n, x = n * p) \wedge (mdc(x,q) = 1)$.

Como já mencionado, esta parte utiliza uma notação mais concisa do que o teorema `ProofCoprime` em dois aspectos. O primeiro é a omissão, quando possível, da definição de hipóteses preliminares em linhas individuais, sendo as premissas necessárias aplicadas, em sucessão, sobre os passos do teorema. Para o leitor que já tiver adquirido um grau de familiaridade com o funcionamento do Lean, esta notação permite um acompanhamento sem digressões do desenvolvimento do teorema.

O segundo aspecto em que o teorema a seguir difere daqueles já apresentados diz respeito à ausência quase integral de rótulos para a identificação dos passos. Isto é possível graças ao comando `this` do Lean. Este comando permite acessar o último passo definido através de um comando `have` e é extremamente útil para desenvolvimento de teoremas sem o "entulhamento" provocado pelos rótulos, sendo necessária a atribuição destes apenas quando uma premissa for ser utilizada em um passo que não seja aquele imediatamente em sucessão ao passo em que foi alcançada.

Por fim, é necessário descrever alguns comandos do Lean que ainda não foram vistos:

PUC-Rio - Certificação Digital Nº 1521382/CA

- Os comandos `and.elim_left` e `and.elim_right` recebem uma conjunção do tipo `a ∧ b` e retornam, respectivamente, uma proposição do tipo `a` ou do tipo `b`.
- Os comandos `iff.elim_left` e `iff.elim_right` recebem uma conjunção do tipo `a ↔ b` e retornam, respectivamente, uma proposição do tipo `a → b` ou do tipo `b → a`.
- Os comandos `mul.comm` e `mul.assoc` chamam as operações de comutatividade e associatividade da multiplicação já definidas na biblioteca do Lean, em forma de um objeto igualdade que pode ser utilizado para realizar substituições com `eq.subst`. Por exemplo, `mul.comm a b` retorna um objeto do tipo `a * b = b * a`, e `mul.assoc a b c` retorna um objeto do tipo `(a * b) * c = a * (b * c)`.

Código	Comentários
`theorem ProofNotCoprime_Part2 (t : nat) : (∃ n, x = n * p) ∧ (gcd x q = one) → congruent (x * ((x ^ (phi n)) ^ t)) x n :=`	
`assume H1 : (∃ n, x = n * p) ∧ (gcd x q = one),`	Hipótese inicial, i.e., $(\exists n, x = n * p) \wedge (mdc(x,q) = 1)$
`have gcd x q = one, from and.elim_right H1,`	Obtenção de prova de $mdc(x,q) = 1$ por eliminação à direita na conjunção lógica acima
`have congruent (x ^ (phi q)) one q, from Euler x q this,`	Prova de $x^{\phi(q)} \equiv 1\ (mod\ q)$ por aplicação do teorema de Euler
`have congruent ((x ^ (phi q)) ^ t) (one ^ t) q, from CongruenceExponentiation (x ^ (phi q)) one q t this,`	Prova de $\left(x^{\phi(q)}\right)^t \equiv 1^t\ (mod\ q)$ por conservação da congruência sob exponenciação com o argumento t
`have congruent ((x ^ (phi q)) ^ t) one q, from eq.subst (OneExp t) this,`	Prova de $\left(x^{\phi(q)}\right)^t \equiv 1\ (mod\ q)$ por cancelamento de expoente sobre base 1
`have congruent (((x ^ (phi q)) ^ t) ^ (p - one)) (one ^ (p - one)) q, from CongruenceExponentiation ((x ^ (phi q)) ^ t) one q (p - one) this,`	Elevação de ambos os lados ao expoente $(p-1)$; prova de $\left(\left(x^{\phi(q)}\right)^t\right)^{(p-1)} \equiv 1^{(p-1)}\ (mod\ q)$

PUC-Rio - Certificação Digital Nº 1521382/CA

`have congruent (((x ^ (phi q)) ^ t) ^ (p - one)) one q,` `  from eq.subst (OneExp (p - one)) this,`	Prova de $\left(\left(x^{\phi(q)}\right)^t\right)^{(p-1)} \equiv 1\ (mod\ q)$ por cancelamento de expoente sobre base 1
`have congruent (((x ^ (phi q)) ^ (p - one)) ^ t) one q,` `  from eq.subst (ExpSwap (x ^ (phi q)) t (p - one)) this,`	Prova de $\left(\left(x^{\phi(q)}\right)^{(p-1)}\right)^t \equiv 1\ (mod\ q)$ por intercambialidade de expoentes
`have congruent ((x ^ ((phi q) * (p - one))) ^ t) one q,` `  from eq.subst (ExpMul x (phi q) (p - one)) this,`	Prova de $\left(x^{\phi(q)*(p-1)}\right)^t \equiv 1\ (mod\ q)$ por multiplicação de expoentes
`have congruent ((x ^ ((q - one) * (p - one))) ^ t) one q,` `  from eq.subst qPhi this,`	Prova de $\left(x^{(q-1)*(p-1)}\right)^t \equiv 1\ (mod\ q)$ por definição de $\phi(q)$ como $q-1$
`have congruent ((x ^ (phi n)) ^ t) one q,` `  from eq.subst (eq.symm nPhi) this,`	Substituição de $(q-1)*(p-1)$ por $\phi(n)$ por definição deste último; obtida prova de $\left(x^{\phi(n)}\right)^t \equiv 1\ (mod\ q)$
`have ∃ c, ((x ^ (phi n)) ^ t) = one + (c * q),` `  from (iff.elim_left (ModuloDef ((x ^ (phi n)) ^ t) one q)) this,`	Obtenção de prova de $\left(x^{\phi(n)}\right)^t \equiv 1\ (mod\ q) \rightarrow \exists c, \left(x^{\phi(n)}\right)^t = 1 + (c * q)$ através da definição de operação de módulo especificada na tabela 5.7, aplicação desta prova sobre o passo anterior para obtenção de prova de $\exists c, \left(x^{\phi(n)}\right)^t = 1 + (c * q)$
`exists.elim this` `(fun (v : nat) (Hv : ((x ^ (phi n)) ^ t) = one + (v * q)),`	Eliminação do quantificador existencial ∃ por introdução de uma nova variável v; prova de $\left(x^{\phi(n)}\right)^t = 1 + (v * q)$
`have x * ((x ^ (phi n)) ^ t) = x * (one + (v * q)),` `  from EqMul ((x ^ (phi n)) ^ t) (one + (v * q)) x Hv,`	Prova de $x * \left(x^{\phi(n)}\right)^t = x * \left(1 + (v * q)\right)$ através de multiplicação por x
`have x * ((x ^ (phi n)) ^ t) = (x * one) + (x * (v * q)),` `  from eq.trans this (MulDistrib x one (v * q)),`	Prova de $x * \left(x^{\phi(n)}\right)^t = (x * 1) + \left(x * (v * q)\right)$ através de distributividade de multiplicação

PUC-Rio - Certificação Digital Nº 1521382/CA

`have x * ((x ^ (phi n)) ^ t) = (one * x) * (x * (v * q)),` `    from eq.subst (mul.comm x one) this,`	Prova de $x * \left(x^{\phi(n)}\right)^t = (1 * x) * \left(x * (v * q)\right)$ por comutatividade sobre $(x * 1)$
`have Hxvq : x * ((x ^ (phi n)) ^ t) = x * (x * (v * q)),` `    from eq.subst (OneMul x) this,`	Prova de $x * \left(x^{\phi(n)}\right)^t = x * \left(x * (v * q)\right)$ por cancelamento de multiplicação por 1. Atribuição do rótulo Hxvq a esta fórmula para uso posterior
`have ∃ n, x = n * p,` `    from and.elim_left H1,`	Obtenção de prova de $\exists n, x = n * p$ por eliminação à esquerda na hipótese original
`exists.elim this` `(fun (w : nat) (Hw : x = w * p),`	Eliminação do quantificador existencial ∃ por introdução de uma nova variável w; prova de $x = w * p$
`have x * ((x ^ (phi n)) ^ t) = x * ((w * p) * (v * q)),` `    from eq.subst Hw Hxvq,`	Substituição de $x = w * p$ em Hxvq; obtida prova de $x * \left(x^{\phi(n)}\right)^t = x * \left((w * p) * (v * q)\right)$
`have x * ((x ^ (phi n)) ^ t) = x * (w * (p * (v * q))),` `    from eq.subst (mul.assoc w p (v * q)) this,`	Prova de $x * \left(x^{\phi(n)}\right)^t = x * \left(w * \left(p * (v * q)\right)\right)$ por associatividade da multiplicação
`have x * ((x ^ (phi n)) ^ t) = x * (w * (p * (q * v))),` `    from eq.subst (mul.comm v q) this,`	Prova de $x * \left(x^{\phi(n)}\right)^t = x * \left(w * \left(p * (q * v)\right)\right)$ por comutatividade da multiplicação
`have x * ((x ^ (phi n)) ^ t) = x * (w * ((p * q) * v)),` `    from eq.subst (eq.symm (mul.assoc p q v)) this,`	Prova de $x * \left(x^{\phi(n)}\right)^t = x * \left(w * \left((p * q) * v\right)\right)$ por associatividade da multiplicação
`have x * ((x ^ (phi n)) ^ t) = x * (w * (n * v)),` `    from eq.subst (eq.symm nDef) this,`	Prova de $x * \left(x^{\phi(n)}\right)^t = x * \left(w * (n * v)\right)$ por definição de n como $p * q$

PUC-Rio - Certificação Digital Nº 1521382/CA

`have x * ((x ^ (phi n)) ^ t) = x + (w * (v * n)),` `  from eq.subst (mul.comm n v) this,`	Prova de $x * \left(x^{\phi(n)}\right)^t = x + \left(w * (v * n)\right)$ por comutatividade
`have x * ((x ^ (phi n)) ^ t) = x + (w * v) * n,` `  from eq.subst (eq.symm (mul.assoc w v n)) this,`	Prova de $x * \left(x^{\phi(n)}\right)^t = x + \left((w * v) * n\right)$ por associatividade
`have ∃ i, x * ((x ^ (phi n)) ^ t) = x + i * n,` `  from exists.intro (w * v) this,`	Introdução de uma variável i igual a $w * v$; obtemos prova de $\exists i, x * \left(x^{\phi(n)}\right)^t = x + (i * n)$

`show congruent (x * ((x ^ (phi n)) ^ t)) x n,`
`  from (iff.elim_right`
`  (ModuloDef (x * ((x ^ (phi n)) ^ t)) x n)) this))`
	Prova de $x \cdot \left(x^{\phi(n)}\right)^t \equiv x\ (mod\ n)$ pela definição de módulo especificada na tabela 5.7

Tabela 5.12: Segunda parte da prova de corretude do funcionamento do algoritmo RSA no Lean Theorem Prover, para o caso em que $mdc(x,n) \neq 1$

Para a outra metade de fórmula inicial, ou $(\exists n, x = n * q) \wedge (mdc(x,p) = 1)$, podemos alcançar a fórmula desejada da mesma forma, apenas trocando as variáveis p e q uma pela outra na descrição da prova.

Acabamos de provar que $x \cdot \left(x^{\phi(n)}\right)^t \equiv x\ (mod\ n)$, tanto para o caso em que $mdc(x,n) \neq 1$ quanto para o caso em que $mdc(x,n) \neq 1$.

Agora, para concluirmos a prova de corretude do RSA, basta mostrarmos como chegar de $x \cdot \left(x^{\phi(n)}\right)^t \equiv x\ (mod\ n)$ a $x^{1+\phi(n)\cdot t} \equiv x\ (mod\ n)$, conforme visto na seção 5.2.1. Vejamos como se faz isto no Lean Theorem Prover:

PUC-Rio - Certificação Digital Nº 1521382/CA

Código	Comentários

`theorem ProofFinal (t : nat) :`
`congruent (x * ((x ^ (phi n)) ^ t)) x n →`
`congruent (x ^ (one + ((phi n) * t))) x n :=`	

`assume H1 : congruent (x * ((x ^ (phi n)) ^ t)) x n,`	Premissa inicial: $x \cdot \left(x^{\phi(n)}\right)^t \equiv x\ (mod\ n)$
`have H2 : congruent (x * (x ^ ((phi n) * t))) x n,` `from eq.subst (ExpMul x (phi n) t) H1,`	Prova de $x \cdot x^{\phi(n)\cdot t} \equiv x\ (mod\ n)$ pela multiplicação de expoentes
`have Honex : x = x ^ one,` `from eq.symm (ExpOne x),`	Prova de $x = x^1$ obtida a partir de ExpOne
`have H3 : congruent ((x ^ one) * (x ^ ((phi n) * t))) x n,` `from eq.subst Honex H2,`	Prova de $x^1 \cdot x^{\phi(n)\cdot t} \equiv x\ (mod\ n)$ a partir da prova de $x = x^1$
`show congruent (x ^ (one + ((phi n) * t))) x n,` `from eq.subst (ExpSum x one ((phi n) * t)) H3`	Prova de $x^{1+\phi(n)\cdot t} \equiv x\ (mod\ n)$ pela soma de expoentes

Tabela 5.13: Parte final da prova de corretude do funcionamento do algoritmo RSA no Lean Theorem Prover

5.3 Provas de Corretude do Algoritmo Blowfish

5.3.1 Prova

Descreve-se aqui a prova de funcionamento do algoritmo Blowfish, visto na seção 3.3. Queremos provar que os dados recuperados após os 16 passos de codificação correspondem aos dados inicias, ou seja:

$$L_0 R_0 = L_{16}^d R_{16}^d$$

Partimos das seguintes premissas, por definição do Blowfish. Primeiro, temos que o primeiro passo da etapa de decodificação é, por definição, o último da etapa de codificação:

$$L_{16} R_{16} = L_0^d R_0^d$$

PUC-Rio - Certificação Digital Nº 1521382/CA

Depois, temos as demais premissas, derivadas da definição do algoritmo dada na seção 3.3:

$$
\left.
\begin{aligned}
&H_{L_1}:\ L_1 = R_0 \oplus F(L_0 \oplus P_1)\\
&H_{R_1}: R_1 = L_0 \oplus P_1\\
&\ldots\\
&H_{L_{15}}: L_{15} = R_1 \oplus F(L_1 \oplus P_{15})\\
&H_{R_{15}}: R_{15} = L_1 \oplus P_{15}
\end{aligned}
\right\}
\text{Definição das premissas } H_{L_i}, H_{R_i} \text{ para } 1 \le i \le 15, \text{ de acordo com a fórmula descrita na seção 3.3}
$$

$$
\begin{aligned}
&H_{L_{16}}: L_{16} = \big(R_{15} \oplus F(L_{15} \oplus P_{16})\big) \oplus P_{17}\\
&H_{R_{16}}: R_{16} = (L_{15} \oplus P_{16}) \oplus P_{18}
\end{aligned}
$$

$$
\left.
\begin{aligned}
&H_{L_1^d}: L_1 = R_0 \oplus F(L_0 \oplus P_1)\\
&H_{R_1^d}: R_1 = L_0 \oplus P_1\\
&\ldots\\
&H_{L_{15}^d}: L_{15} = R_1 \oplus F(L_1 \oplus P_{15})\\
&H_{R_{15}^d}: R_{15} = L_1 \oplus P_{15}
\end{aligned}
\right\}
\text{Definição das premissas } H_{L_i^d}, H_{R_i^d} \text{ para } 1 \le i \le 15, \text{ de acordo com a fórmula descrita na seção 3.3}
$$

$$
\begin{aligned}
&H_{L_{16}^d}: L_{16} = \big(R_{15} \oplus F(L_{15} \oplus P_{16})\big) \oplus P_{17}\\
&H_{R_{16}^d}: R_{16} = (L_{15} \oplus P_{16}) \oplus P_{18}
\end{aligned}
$$

5.3.2 Prova no Lean Theorem Prover

Vejamos agora como definir no Lean todas as variáveis e premissas necessárias para realizar a prova de corretude do Blowfish.

Primeiro, definimos as variáveis que representarão os blocos de dados e as chaves, assim como as operações $\oplus$ (ou exclusivo) e F:

```
variable data : Type

variables
L0  L1  L2  L3  L4  L5  L6  L7  L8
L9  L10 L11 L12 L13 L14 L15 L16

R0  R1  R2  R3  R4  R5  R6  R7  R8
R9  R10 R11 R12 R13 R14 R15 R16

L0_D  L1_D  L2_D  L3_D  L4_D  L5_D  L6_D  L7_D  L8_D
L9_D  L10_D L11_D L12_D L13_D L14_D L15_D L16_D

R0_D  R1_D  R2_D  R3_D  R4_D  R5_D  R6_D  R7_D  R8_D
R9_D  R10_D R11_D R12_D R13_D R14_D R15_D R16_D
```

PUC-Rio - Certificação Digital Nº 1521382/CA

```
P1  P2  P3  P4  P5  P6  P7  P8  P9
P10 P11 P12 P13 P14 P15 P16 P17 P18        : data

variable xor (d1 d2 : data) : data
variable F (d : data) : data
```

Em seguida, definimos as propriedades de $\oplus$:

Código no Lean Theorem Prover	Propriedade
`premise xor_comm  (x1 x2 : data) : (xor x1 x2) = (xor x2 x1)`	$a \oplus b = b \oplus a$
`premise xor_assoc (x1 x2 x3 : data) : (xor (xor x1 x2) x3) = (xor x1 (xor x2 x3))`	$(a \oplus b) \oplus c = a \oplus (b \oplus c)$
`premise xor_cancel : ∀ x1 x2 : data, xor x1 (xor x2 x2) = x1`	$a \oplus (b \oplus b) = a$
`premise xor_swap : ∀ x1 x2 x3 : data, x1 = xor x2 x3 → x2 = xor x1 x3`	Se $a = b \oplus c$, então $b = a \oplus c$

Tabela 5.14: Definição no Lean das propriedades da operação de ou-exclusivo para a prova de corretude do algoritmo Blowfish

Agora, devemos definir cada passo das etapas de codificação e decodificação em função dos passos anteriores. Por exemplo, as definições de L_1 e R_1, de acordo com o algoritmo Blowfish, são:

$L_1 = R_0 \oplus F(L_0 \oplus P_1)$
$R_1 = L_0 \oplus P_1$.

Descrevemos estas definições no Lean da seguinte forma:

```
premise L1_Def        : L1  = xor R0  (F (xor L0 P1))
premise R1_Def        : R1  = xor L0  P1
```

Continuamos realizando a definição dos demais passos da fase de codificação, L_2R_2 a $L_{16}R_{16}$, conforme as premissas vistas na seção 5.3.1:

```
premise L2_Def        : L2  = xor R1  (F (xor L1 P2))
premise R2_Def        : R2  = xor L1  P2
premise L3_Def        : L3  = xor R2  (F (xor L2 P3))
premise R3_Def        : R3  = xor L2  P3
premise L4_Def        : L4  = xor R3  (F (xor L3 P4))
premise R4_Def        : R4  = xor L3  P4
premise L5_Def        : L5  = xor R4  (F (xor L4 P5))
```

PUC-Rio - Certificação Digital Nº 1521382/CA

```
premise R5_Def        : R5  = xor L4  P5
premise L6_Def        : L6  = xor R5  (F (xor L5 P6))
premise R6_Def        : R6  = xor L5  P6
premise L7_Def        : L7  = xor R6  (F (xor L6 P7))
premise R7_Def        : R7  = xor L6  P7
premise L8_Def        : L8  = xor R7  (F (xor L7 P8))
premise R8_Def        : R8  = xor L7  P8
premise L9_Def        : L9  = xor R8  (F (xor L8 P9))
premise R9_Def        : R9  = xor L8  P9
premise L10_Def       : L10 = xor R9  (F (xor L9 P10))
premise R10_Def       : R10 = xor L9  P10
premise L11_Def       : L11 = xor R10 (F (xor L10 P11))
premise R11_Def       : R11 = xor L10 P11
premise L12_Def       : L12 = xor R11 (F (xor L11 P12))
premise R12_Def       : R12 = xor L11 P12
premise L13_Def       : L13 = xor R12 (F (xor L12 P13))
premise R13_Def       : R13 = xor L12 P13
premise L14_Def       : L14 = xor R13 (F (xor L13 P14))
premise R14_Def       : R14 = xor L13 P14
premise L15_Def       : L15 = xor R14 (F (xor L14 P15))
premise R15_Def       : R15 = xor L14 P15
premise L16_Def       : L16 = xor P18 (xor L15 P16)
premise R16_Def       : R16 = xor P17 (xor R15 (F (xor L15 P16)))
```

Por fim, definimos os passos da fase de decodificação, de $L_0^d R_0^d$ a $L_{16}^d R_{16}^d$:

```
premise L0_D_Def      : L0_D  = L16
premise R0_D_Def      : R0_D  = R16
premise L1_D_Def      : L1_D  = xor R0_D  (F (xor L0_D P18))
premise R1_D_Def      : R1_D  = xor L0_D  P18
premise L2_D_Def      : L2_D  = xor R1_D  (F (xor L1_D P17))
premise R2_D_Def      : R2_D  = xor L1_D  P17
premise L3_D_Def      : L3_D  = xor R2_D  (F (xor L2_D P16))
premise R3_D_Def      : R3_D  = xor L2_D  P16
premise L4_D_Def      : L4_D  = xor R3_D  (F (xor L3_D P15))
premise R4_D_Def      : R4_D  = xor L3_D  P15
premise L5_D_Def      : L5_D  = xor R4_D  (F (xor L4_D P14))
premise R5_D_Def      : R5_D  = xor L4_D  P14
premise L6_D_Def      : L6_D  = xor R5_D  (F (xor L5_D P13))
premise R6_D_Def      : R6_D  = xor L5_D  P13
premise L7_D_Def      : L7_D  = xor R6_D  (F (xor L6_D P12))
premise R7_D_Def      : R7_D  = xor L6_D  P12
premise L8_D_Def      : L8_D  = xor R7_D  (F (xor L7_D P11))
premise R8_D_Def      : R8_D  = xor L7_D  P11
premise L9_D_Def      : L9_D  = xor R8_D  (F (xor L8_D P10))
premise R9_D_Def      : R9_D  = xor L8_D  P10
premise L10_D_Def     : L10_D = xor R9_D  (F (xor L9_D P9))
premise R10_D_Def     : R10_D = xor L9_D  P9
premise L11_D_Def     : L11_D = xor R10_D (F (xor L10_D P8))
premise R11_D_Def     : R11_D = xor L10_D P8
premise L12_D_Def     : L12_D = xor R11_D (F (xor L11_D P7))
premise R12_D_Def     : R12_D = xor L11_D P7
```

PUC-Rio - Certificação Digital Nº 1521382/CA

```
premise L13_D_Def     : L13_D = xor R12_D (F (xor L12_D P6))
premise R13_D_Def     : R13_D = xor L12_D P6
premise L14_D_Def     : L14_D = xor R13_D (F (xor L13_D P5))
premise R14_D_Def     : R14_D = xor L13_D P5
premise L15_D_Def     : L15_D = xor R14_D (F (xor L14_D P4))
premise R15_D_Def     : R15_D = xor L14_D P4
premise L16_D_Def     : L16_D = xor P1 (xor L15_D P3)
premise R16_D_Def     : R16_D = xor P2 (xor R15_D (F (xor L15_D
P3)))
```

Agora, com base nas variáveis e premissas que acabamos de definir, podemos descrever no Lean a prova de corretude do algoritmo Blowfish, vista abaixo:

Código	Comentários
`theorem ProofL1 : L1_D = xor P17 R15, :=`	Primeira etapa. Aqui, visamos provar que $\boldsymbol{L_1^d = P_{17} \oplus R_{15}}$
`have H1 : L1_D = xor R0_D (F (xor L0_D P18)), from L1_D_Def,`	Prova de $L_1^d = R_0^d \oplus F(L_0^d \oplus P_{18})$ pela definição de L_1^d
`have H2 : L1_D = xor R16 (F (xor L0_D P18)),` `from eq.subst R0_D_Def H1,`	Prova de $L_1^d = R_{16} \oplus F(L_0^d \oplus P_{18})$ pela definição de R_0^d
`have H3 : L1_D = xor R16 (F (xor L16 P18)),` `from eq.subst L0_D_Def H2,`	Prova de $L_1^d = R_{16} \oplus F(L_{16} \oplus P_{18})$ pela definição de L_0^d
`have H4 : L1_D = xor R16 (F (xor (xor P18 (xor L15 P16)) P18)),` `from eq.subst L16_Def H3,`	Prova de $L_1^d = R_{16} \oplus F\left(\left(P_{18} \oplus (L_{15} \oplus P_{16})\right) \oplus P_{18}\right)$ pela definição de L_{16}
`have H5 : L1_D = xor R16 (F (xor (xor (xor L15 P16) P18) P18)),` `from eq.subst (xor_comm P18 (xor L15 P16)) H4,`	Prova de $L_1^d = R_{16} \oplus F\left(\left((L_{15} \oplus P_{16}) \oplus P_{18}\right) \oplus P_{18}\right)$ pela comutatividade de $\oplus$
`have H6 : L1_D = xor R16 (F (xor (xor L15 P16) (xor P18 P18))),` `from eq.subst (xor_assoc (xor L15 P16) P18 P18) H5,`	Prova de $L_1^d = R_{16} \oplus F\left((L_{15} \oplus P_{16}) \oplus (P_{18} \oplus P_{18})\right)$ pela associatividade de $\oplus$

PUC-Rio - Certificação Digital Nº 1521382/CA

PUC-Rio - Certificação Digital Nº 1521382/CA

`have H7 : L1_D = xor R16 (F (xor L15 P16)),` `  from eq.subst (xor_cancel (xor L15 P16) P18) H6,`	Prova de $L_1^d = R_{16} \oplus F(L_{15} \oplus P_{16})$ pela idempotência de $\oplus$

`have H8 : L1_D = xor (xor P17 (xor R15 (F (xor L15 P16))))`
`  (F (xor L15 P16)),`
`  from eq.subst R16_Def H7,`
	Prova de $L_1^d = \left(\begin{matrix} P_{17} \oplus \\ (R_{15} \oplus F(L_{15} \oplus P_{16})) \end{matrix}\right) \oplus F(L_{15} \oplus P_{16})$ pela definição de R_{16}

`have H9 : L1_D = xor P17 (xor (xor R15 (F (xor L15 P16)))`
`  (F (xor L15 P16)) ),`
`  from eq.subst (xor_assoc P17`
`  (xor R15 (F (xor L15 P16)))`
`  ((F (xor L15 P16)))`
`  ) H8,`
	Prova de $L_1^d = P_{17} \oplus \left(\begin{matrix} (R_{15} \oplus F(L_{15} \oplus P_{16})) \\ \oplus F(L_{15} \oplus P_{16}) \end{matrix}\right)$ pela associatividade de $\oplus$

`have H10 : L1_D = xor P17 (xor R15 (xor (F (xor L15 P16))`
`  (F (xor L15 P16)))),`
`  from eq.subst (xor_assoc R15`
`  (F (xor L15 P16))`
`  (F (xor L15 P16))`
`  ) H9,`
	Prova de $L_1^d = P_{17} \oplus \left(R_{15} \oplus \left(\begin{matrix} F(L_{15} \oplus P_{16}) \\ \oplus F(L_{15} \oplus P_{16}) \end{matrix}\right)\right)$ pela associatividade de $\oplus$

`show L1_D = xor P17 R15,` `  from eq.subst (xor_cancel R15 (F (xor L15 P16))) H10`	Prova de $\boldsymbol{L_1^d = P_{17} \oplus R_{15}}$ por cancelamento de $\oplus$. Esta equação será utilizada adiante; a denominamos $\boldsymbol{Eq1}$.

`theorem ProofR1 : R1_D = xor L15 P16 :=`	Próxima etapa. Agora, visamos provar que $\boldsymbol{R_1^d = L_{15} \oplus P_{16}}$
`have H1 : R1_D = xor L0_D P18,` `  from R1_D_Def,`	Prova de $R_1^d = L_0^d \oplus P_{18}$ pela definição de R_1^d
`have H2 : R1_D = xor L16 P18,` `  from eq.subst L0_D_Def H1,`	Prova de $R_1^d = L_{16} \oplus P_{18}$ pela definição de L_0^d
`have H3 : R1_D = xor (xor P18 (xor L15 P16)) P18,` `  from eq.subst L16_Def H2,`	Prova de $R_1^d = (P_{18} \oplus (L_{15} \oplus P_{16})) \oplus P_{18}$ pela definição de L_{16}

PUC-Rio - Certificação Digital Nº 1521382/CA

`have H4 : R1_D = xor (xor (xor L15 P16) P18) P18,` `  from eq.subst (xor_comm P18 (xor L15 P16)) H3,`	Prova de $R_1^d = \big((L_{15} \oplus P_{16}) \oplus P_{18}\big) \oplus P_{18}$ por comutatividade de $\oplus$
`have H5 : R1_D = xor (xor L15 P16) (xor P18 P18),` `  from eq.subst (xor_assoc (xor L15 P16) P18 P18) H4,`	Prova de $R_1^d = (L_{15} \oplus P_{16}) \oplus (P_{18} \oplus P_{18})$ por associatividade de $\oplus$
`show R1_D = xor L15 P16,` `  from eq.subst (xor_cancel (xor L15 P16) P18) H6`	Prova de $\boldsymbol{R_1^d = L_{15} \oplus P_{16}}$ por cancelamento de $\oplus$. Denominamos esta equação $\boldsymbol{Eq2}$.

`theorem ProofStep2`
`(S1_L : L1_D = xor P17 R15) (S1_R : R1_D = xor L15 P16) :`
`(L2_D = xor (xor L15 P16) (F R15)) ∧ (R2_D = R15) :=`
	Próxima etapa. Agora, partimos de $Eq1$ e $Eq2$ e temos como objetivo provar que $\boldsymbol{L_2^d = (L_{15} \oplus P_{16}) \oplus F(R_{15})}$ e $\boldsymbol{R_2^d = R_{15}}$.

`have H1 : R2_D = xor L1_D P17,` `  from R2_D_Def,`	Prova de $R_2^d = L_1^d \oplus P_{17}$ pela definição de R_2^d. Esta equação também será usada mais tarde; vamos chamá-la de $\boldsymbol{Eq3}$.
`have H2 : R2_D = xor (xor P17 R15) P17,` `  from eq.subst S1_L H1,`	Prova de $R_2^d = (P_{17} \oplus R_{15}) \oplus P_{17}$ por substituição de $Eq1$
`have H3 : R2_D = xor (xor R15 P17) P17,` `  from eq.subst (xor_comm P17 R15) H2,`	Prova de $R_2^d = (R_{15} \oplus P_{17}) \oplus P_{17}$ por comutatividade de $\oplus$
`have H4 : R2_D = xor R15 (xor P17 P17),` `  from eq.subst (xor_assoc R15 P17 P17) H3,`	Prova de $R_2^d = R_{15} \oplus (P_{17} \oplus P_{17})$ por associatividade de $\oplus$
`have H5 : R2_D = R15,` `  from eq.subst (xor_cancel R15 P17) H4,`	Prova de $\boldsymbol{R_2^d = R_{15}}$ por idempotência de $\oplus$. Denominamos esta equação $\boldsymbol{Eq4}$.
`have H6 : L2_D = xor R1_D (F (xor L1_D P17)),` `  from L2_D_Def,`	Prova de $L_2^d = R_1^d \oplus F(L_1^d \oplus P_{17})$ pela definição de L_2^d
`have H7 : L2_D = xor R1_D (F R2_D),` `  from eq.subst (eq.symm H1) H6,`	Prova de $L_2^d = R_1^d \oplus F(R_2^d)$ por substituição de $Eq3$

PUC-Rio - Certificação Digital Nº 1521382/CA

have H8 : L2_D = xor R1_D (F R15), from eq.subst H5 H7,	Prova de $L_2^d = R_1^d \oplus F(R_{15})$ por substituição de $Eq4$
have H9 : L2_D = xor (xor L15 P16) (F R15), from eq.subst S1_R H8,	Prova de $\boldsymbol{L_2^d = (L_{15} \oplus P_{16}) \oplus F(R_{15})}$ por substituição de $Eq2$. Denominamos esta equação $\boldsymbol{Eq5}$.
show (L2_D = xor (xor L15 P16) (F R15)) ∧ (R2_D = R15), from and.intro H9 H5	Conjunção lógica das duas provas que obtivemos ($Eq5$ e $Eq4$); esta parte tem apenas a finalidade simbólica de representar completamente o segundo passo da prova de corretude do Blowfish
	---
theorem ProofStep3 (S2_L : L2_D = xor (xor L15 P16) (F R15)) (S2_R : R2_D = R15) : (L3_D = xor R13 P15) ∧ (R3_D = R14) :=	Próxima etapa. Aqui, provaremos que $\boldsymbol{L_3^d = R_{13} \oplus P_{15}}$ e $\boldsymbol{R_3^d = R_{14}}$.
have H1 : R3_D = xor L2_D P16, from R3_D_Def,	Prova de $R_3^d = L_2^d \oplus P_{16}$ pela definição de R_3^d
have H2 : R3_D = xor (xor (xor L15 P16) (F R15)) P16, from eq.subst S2_L H1,	Prova de $R_3^d = \big((L_{15} \oplus P_{16}) \oplus F(R_{15})\big) \oplus P_{16}$ por substituição de $Eq5$
have H3 : R3_D = xor (xor (F R15) (xor L15 P16)) P16, from eq.subst (xor_comm (xor L15 P16) (F R15)) H2,	Prova de $R_3^d = \big(F(R_{15}) \oplus (L_{15} \oplus P_{16})\big) \oplus P_{16}$ por comutatividade de $\oplus$
have H4 : R3_D = xor (F R15) (xor (xor L15 P16) P16), from eq.subst (xor_assoc (F R15) (xor L15 P16) P16) H3,	Prova de $R_3^d = F(R_{15}) \oplus \big((L_{15} \oplus P_{16}) \oplus P_{16}\big)$ por associatividade de $\oplus$
have H5 : R3_D = xor (F R15) (xor L15 (xor P16 P16)), from eq.subst (xor_assoc L15 P16 P16) H4,	Prova de $R_3^d = F(R_{15}) \oplus \big(L_{15} \oplus (P_{16} \oplus P_{16})\big)$ por associatividade de $\oplus$
have H6 : R3_D = xor (F R15) L15, from eq.subst (xor_cancel L15 P16) H5,	Prova de $R_3^d = F(R_{15}) \oplus L_{15}$ por idempotência de $\oplus$. Denominamos esta equação $\boldsymbol{Eq6}$.
have H7 : L15 = xor R14 (F R15), from eq.subst (eq.symm R15_Def) L15_Def,	Prova de $L_{15} = R_{14} \oplus F(R_{15})$ pelas definições de L_{15} e R_{15}

`have H8 : R14 = xor L15 (F R15), from (xor_swap L15 R14 (F R15)) H7,`	Prova de $R_{14} = L_{15} \oplus F(R_{15})$ através da troca de $\oplus$
`have H9 : R14 = xor (F R15) L15, from eq.subst (xor_comm L15 (F R15)) H8,`	Prova de $R_{14} = F(R_{15}) \oplus L_{15}$ por comutatividade de $\oplus$. Denominamos esta equação $\boldsymbol{Eq7}$.
`have H10 : R3_D = R14, from eq.trans H6 (eq.symm H9),`	Prova de $R_3^d = R_{14}$ por aplicação da transitividade sobre $Eq6$ e $Eq7$. Denominamos esta equação $\boldsymbol{Eq8}$.
`have H11 : L3_D = xor R15 (F (xor L2_D P16)), from eq.subst S2_R L3_D_Def,`	Prova de $L_3^d = R_{15} \oplus F(L_2^d \oplus P_{16})$ por substituição de $Eq4$ na definição de L_3^d
`have H12 : L3_D = xor R15 (F R3_D), from eq.subst (eq.symm H1) H11,`	Prova de $L_3^d = R_{15} \oplus F(R_3^d)$ pela definição de R_3^d
`have H13 : L3_D = xor R15 (F R14), from eq.subst H10 H12,`	Prova de $L_3^d = R_{15} \oplus F(R_{14})$ por substituição de $Eq8$
`have H14 : L3_D = xor (xor L14 P15) (F R14), from eq.subst R15_Def H13,`	Prova de $L_3^d = (L_{14} \oplus P_{15}) \oplus F(R_{14})$ por definição de R_{15}
`have H15 : L3_D = xor (xor L14 P15) (F (xor L13 P14)), from eq.subst R14_Def H14,`	Prova de $L_3^d = (L_{14} \oplus P_{15}) \oplus F(L_{13} \oplus P_{14})$ por definição de R_{14}
`have H16 : L3_D = xor (xor P15 L14) (F (xor L13 P14)), from eq.subst (xor_comm L14 P15) H15,`	Prova de $L_3^d = (P_{15} \oplus L_{14}) \oplus F(L_{13} \oplus P_{14})$ por comutatividade de $\oplus$
`have H17 : L3_D = xor P15 (xor L14 (F (xor L13 P14))), from eq.subst (xor_assoc P15 L14 (F (xor L13 P14))) H16,`	Prova de $L_3^d = P_{15} \oplus \big(L_{14} \oplus F(L_{13} \oplus P_{14})\big)$ por associatividade de $\oplus$. Denominamos esta equação $\boldsymbol{Eq9}$.
`have H18 : R13 = xor L14 (F (xor L13 P14)), from (xor_swap L14 R13 (F (xor L13 P14))) L14_Def,`	Prova de $R_{13} = L_{14} \oplus F(L_{13} \oplus P_{14})$ através da troca de $\oplus$ na definição de L_{14}. Denominamos esta equação $\boldsymbol{Eq10}$.
`have H19 : L3_D = xor P15 R13, from eq.subst (eq.symm H18) H17,`	Prova de $L_3^d = P_{15} \oplus R_{13}$ pela substituição de $Eq10$ em $Eq9$
`have H20 : L3_D = xor R13 P15, from eq.subst (xor_comm P15 R13) H19,`	Prova de $L_3^d = R_{13} \oplus P_{15}$ por comutatividade de $\oplus$. Denominamos esta equação $\boldsymbol{Eq11}$.

PUC-Rio - Certificação Digital Nº 1521382/CA

`show (L3_D = xor R13 P15) ∧ (R3_D = R14),` `  from and.intro H20 H10`	Conjunção lógica das duas provas que obtivemos: $\boldsymbol{L_3^d = R_{13} \oplus P_{15}}$ e $\boldsymbol{R_3^d = R_{14}}$.

`theorem ProofStep4`
`(S3_R : L3_D = xor R13 P15) (S3_L : R3_D = R14) :`
`(L4_D = xor R12 P14) ∧ (R4_D = R13) :=`
	Próxima etapa. Aqui, provaremos que $\boldsymbol{L_4^d = R_{12} \oplus P_{14}}$ e $\boldsymbol{R_4^d = R_{13}}$.

`have H1 : R4_D = xor L3_D P15,` `  from R4_D_Def,`	Prova de $R_4^d = L_3^d \oplus P_{15}$ pela definição de R_4^d
`have H2 : R4_D = xor (xor R13 P15) P15,` `  from eq.subst S3_R H1,`	Prova de $R_4^d = (R_{13} \oplus P_{15}) \oplus P_{15}$ por substituição de $Eq11$
`have H3 : R4_D = xor R13 (xor P15 P15),` `  from eq.subst (xor_assoc R13 P15 P15) H2,`	Prova de $R_4^d = R_{13} \oplus (P_{15} \oplus P_{15})$ por associatividade de $\oplus$
`have H5 : R4_D = R13,` `  from eq.subst (xor_cancel R13 P15) H3,`	Prova de $R_4^d = R_{13}$ por cancelamento de $\oplus$
`have H6 : R13 = xor L3_D P15,` `  from eq.trans (eq.symm H5) H1,`	Prova de $R_{13} = L_3^d \oplus P_{15}$ por transitividade com a definição de R_4^d. Denominamos esta equação $\boldsymbol{Eq12}$.
`have H7 : L4_D = xor R3_D (F (xor L3_D P15)),` `  from L4_D_Def,`	Prova de $L_4^d = R_3^d \oplus F(L_3^d \oplus P_{15})$ por definição de L_4^d
`have H8 : L4_D = xor R14 (F (xor L3_D P15)),` `  from eq.subst S3_L H7,`	Prova de $L_4^d = R_{14} \oplus F(L_3^d \oplus P_{15})$ por substituição de $Eq8$. Denominamos esta equação $\boldsymbol{Eq13}$.
`have H9 : L4_D = xor R14 (F R13),` `  from eq.subst (eq.symm H6) H8,`	Prova de $L_4^d = R_{14} \oplus F(R_{13})$ pela substituição de $Eq12$ em $Eq13$
`have H10 : L4_D = xor (xor L13 P14) (F R13),` `  from eq.subst R14_Def H9,`	Prova de $L_4^d = (L_{13} \oplus P_{14}) \oplus F(R_{13})$ por substituição da definição de R_{14}
`have H11 : L4_D = xor (xor P14 L13) (F R13),` `  from eq.subst (xor_comm L13 P14) H10,`	Prova de $L_4^d = (P_{14} \oplus L_{13}) \oplus F(R_{13})$ por comutatividade de $\oplus$
`have H12 : L4_D = xor P14 (xor L13 (F R13)),` `  from eq.subst (xor_assoc P14 L13 (F R13)) H11,`	Prova de $L_4^d = P_{14} \oplus \big(L_{13} \oplus F(R_{13})\big)$ por associatividade de $\oplus$

PUC-Rio - Certificação Digital Nº 1521382/CA

`have H13 : L4_D = xor P14 (xor (xor R12 (F (xor L12 P13))) (F R13)),` `  from eq.subst L13_Def H12,`	Prova de $L_4^d = P_{14} \oplus \left(\left(R_{12} \oplus F(L_{12} \oplus P_{13})\right) \oplus F(R_{13})\right)$ por substituição da definição de L_{13}
`have H14 : L4_D = xor P14 (xor R12 (xor (F (xor L12 P13)) (F R13))),` `  from eq.subst (xor_assoc R12 (F (xor L12 P13)) (F R13)) H13,`	Prova de $L_4^d = P_{14} \oplus \left(R_{12} \oplus \left(F(L_{12} \oplus P_{13}) \oplus F(R_{13})\right)\right)$ por associatividade de $\oplus$
`have H15 : L4_D = xor P14 (xor R12 (xor (F (xor L12 P13)) (F (xor L12 P13)))),` `  from eq.subst R13_Def H14,`	Prova de $L_4^d = P_{14} \oplus \left(R_{12} \oplus \left(F(L_{12} \oplus P_{13}) \oplus F(L_{12} \oplus P_{13})\right)\right)$ por substituição da definição de R_{13}
`have H16 : L4_D = xor P14 R12,` `  from eq.subst (xor_cancel R12 (F (xor L12 P13))) H15,`	Prova de $L_4^d = P_{14} \oplus R_{12}$ por cancelamento de $\oplus$
`have H17 : L4_D = xor R12 P14,` `  from eq.subst (xor_comm P14 R12) H16,`	Prova de $L_4^d = R_{12} \oplus P_{14}$ por comutatividade de $\oplus$
`show  (L4_D = xor R12 P14) ∧ (R4_D = R13),` `  from and.intro H17 H5`	Conjunção lógica das duas provas que obtivemos: $\boldsymbol{L_4^d = R_{12} \oplus P_{14}}$ e $\boldsymbol{R_4^d = R_{13}}$.

Tabela 5.15: Prova de corretude do funcionamento do algoritmo Blowfish

Acabamos de provar que $L_3^d = R_{13} \oplus P_{15}$ e $R_3^d = R_{14}$ implicam em $L_4^d = R_{12} \oplus P_{14}$ e $R_4^d = R_{13}$. A partir disto, podemos provar por indução que $L_{15}^d = R_1 \oplus P_3$ e $R_{15}^d = R_2$, realizando provas análogas sucessivamente.

Uma vez tendo obtido a prova de $L_{15}^d = R_1 \oplus P_3$ e $R_{15}^d = R_2$, podemos, enfim, partir para a etapa final da prova de corretude do Blowfish, isto é, a prova de que $L_{16}^d = L_0$ e $R_{16}^d = R_0$.

Número	Equação

`theorem ProofStep5Left`
`(S3_R : L15_D = xor R1 P3) (S3_L : R15_D = R2) :`
`L16_D = L0 :=`
	Próxima etapa. Aqui, provaremos que $\boldsymbol{L_{16}^d = L_0}$.

PUC-Rio - Certificação Digital Nº 1521382/CA

PUC-Rio - Certificação Digital Nº 1521382/CA

`have H1 : L16_D = xor P1 (xor L15_D P3),` `  from L16_D_Def,`	Prova de $L_{16}^{d} = P_1 \oplus (L_{15}^{d} \oplus P_3)$ pela definição de L_{16}^{d}
`have H2 : L16_D = xor P1 (xor (xor R1 P3) P3),` `  from eq.subst S3_R H1,`	Prova de $L_{16}^{d} = P_1 \oplus ((R_1 \oplus P_3) \oplus P_3)$ pela substituição de $L_{15}^{d} = R_1 \oplus P_3$
`have H3 : L16_D = xor P1 (xor R1 (xor P3 P3)),` `  from eq.subst (xor_assoc R1 P3 P3) H2,`	Prova de $L_{16}^{d} = P_1 \oplus (R_1 \oplus (P_3 \oplus P_3))$ por associatividade de $\oplus$
`have H4 : L16_D = xor P1 R1,` `  from eq.subst (xor_cancel R1 P3) H3,`	Prova de $L_{16}^{d} = P_1 \oplus R_1$ por cancelamento de $\oplus$
`have H5 : L16_D = xor P1 (xor L0 P1),` `  from eq.subst R1_Def H4,`	Prova de $L_{16}^{d} = P_1 \oplus (L_0 \oplus P_1)$ pela definição de R_1
`have H6 : L16_D = xor (xor L0 P1) P1,` `  from eq.subst (xor_comm P1 (xor L0 P1)) H5,`	Prova de $L_{16}^{d} = (L_0 \oplus P_1) \oplus P_1$ por comutatividade de $\oplus$
`have H7 : L16_D = xor L0 (xor P1 P1),` `  from eq.subst (xor_assoc L0 P1 P1) H6,`	Prova de $L_{16}^{d} = L_0 \oplus (P_1 \oplus P_1)$ por associatividade de $\oplus$
`show  L16_D = L0,` `  from eq.subst (xor_cancel L0 P1) H7`	Prova de $L_{16}^{d} = L_0$ por cancelamento de $\oplus$

`theorem ProofStep5Right`
`(S3_R : L15_D = xor R1 P3) (S3_L : R15_D = R2) :`
`R16_D = R0 :=`
	Última etapa. Aqui, provaremos que $\boldsymbol{R_{16}^{d} = R_0}$.

`have H1 : R16_D = xor P2 (xor R15_D (F (xor L15_D P3))),` `  from R16_D_Def,`	Prova de $R_{16}^{d} = P_2 \oplus \left(R_{15}^{d} \oplus F\left(L_{15}^{d} \oplus P_3\right)\right)$ pela definição de R_{16}^{d}
`have H2 : R16_D = xor P2 (xor R15_D (F (xor (xor R1 P3) P3))),` `  from eq.subst S3_R H1,`	Prova de $R_{16}^{d} = P_2 \oplus \left(R_{15}^{d} \oplus F\left((R_1 \oplus P_3) \oplus P_3\right)\right)$ pela substituição de $L_{15}^{d} = R_1 \oplus P_3$
`have H3 : R16_D = xor P2 (xor R15_D (F (xor R1 (xor P3 P3)))),` `  from eq.subst (xor_assoc R1 P3 P3) H2,`	Prova de $R_{16}^{d} = P_2 \oplus \left(R_{15}^{d} \oplus F\left(R_1 \oplus (P_3 \oplus P_3)\right)\right)$ por associatividade de $\oplus$

PUC-Rio - Certificação Digital Nº 1521382/CA

`have H4 : R16_D = xor P2 (xor R15_D (F R1)),` `  from eq.subst (xor_cancel R1 P3) H3,`	Prova de $R_{16}^d = P_2 \oplus \left(R_{15}^d \oplus F(R_1)\right)$ por cancelamento de $\oplus$
`have H5 : R16_D = xor P2 (xor R2 (F R1)),` `  from eq.subst S3_L H4,`	Prova de $R_{16}^d = P_2 \oplus \left(R_2 \oplus F(R_1)\right)$ pela substituição de $R_{15}^d = R_2$
`have H6 : R16_D = xor P2 (xor R2 (F (xor L0 P1))),` `  from eq.subst R1_Def H5,`	Prova de $R_{16}^d = P_2 \oplus \left(R_2 \oplus F\left((L_0 \oplus P_1)\right)\right)$ por definição de R_1
`have H7 : R16_D = xor P2 (xor (xor L1 P2) (F (xor L0 P1))),` `  from eq.subst R2_Def H6,`	Prova de $R_{16}^d = P_2 \oplus \left((L_1 \oplus P_2) \oplus F\left((L_0 \oplus P_1)\right)\right)$ por definição de R_2
`have H8 : R16_D = xor P2 (xor (xor P2 L1) (F (xor L0 P1))),` `  from eq.subst (xor_comm L1 P2) H7,`	Prova de $R_{16}^d = P_2 \oplus \left((P_2 \oplus L_1) \oplus F\left((L_0 \oplus P_1)\right)\right)$ por comutatividade de $\oplus$
`have H9 : R16_D = xor P2 (xor P2 (xor L1 (F (xor L0 P1)))),` `  from eq.subst (xor_assoc P2 L1 (F (xor L0 P1))) H8,`	Prova de $R_{16}^d = P_2 \oplus \left(P_2 \oplus \left(L_1 \oplus F\left((L_0 \oplus P_1)\right)\right)\right)$ por associatividade de $\oplus$

`have H10 : R16_D = xor (xor P2 P2) (xor L1 (F (xor L0 P1))),`
`  from eq.subst`
`  (eq.symm (xor_assoc P2 P2 (xor L1 (F (xor L0 P1))))) H9,`
	Prova de $R_{16}^d = (P_2 \oplus P_2) \oplus \left(L_1 \oplus F\left((L_0 \oplus P_1)\right)\right)$ por associatividade de $\oplus$

`have H11 : R16_D = xor (xor L1 (F (xor L0 P1))) (xor P2 P2),`
`  from eq.subst`
`  (xor_comm (xor P2 P2) (xor L1 (F (xor L0 P1)))) H10,`
	Prova de $R_{16}^d = \left(L_1 \oplus F\left((L_0 \oplus P_1)\right)\right) \oplus (P_2 \oplus P_2)$ por comutatividade de $\oplus$

`have H12 : R16_D = (xor L1 (F (xor L0 P1))),`
`  from eq.subst`
`  (xor_cancel (xor L1 (F (xor L0 P1))) P2) H11,`
	Prova de $R_{16}^d = \left(L_1 \oplus F\left((L_0 \oplus P_1)\right)\right)$ por cancelamento de $\oplus$

`have H13 : R16_D = (xor (xor R0 (F (xor L0 P1))) (F (xor L0 P1))),` `  from eq.subst L1_Def H12,`	Prova de $R_{16}^d = \left(\begin{matrix}\left(R_0 \oplus F\left((L_0 \oplus P_1)\right)\right) \\ \oplus F\left((L_0 \oplus P_1)\right)\end{matrix}\right)$ por definição de L_1
`have H14 : R16_D = (xor R0 (xor (F (xor L0 P1)) (F (xor L0 P1)))),` `  from eq.subst (xor_assoc R0 (F (xor L0 P1)) (F (xor L0 P1))) H13,`	Prova de $R_{16}^d = \left(\begin{matrix}R_0 \oplus \\ \left(\begin{matrix}F\left((L_0 \oplus P_1)\right) \oplus \\ F\left((L_0 \oplus P_1)\right)\end{matrix}\right)\end{matrix}\right)$ por associatividade de $\oplus$
`show  R16_D = R0,` `  from eq.subst (xor_cancel R0 (F (xor L0 P1))) H14`	Prova de $R_{16}^d = R_0$ por cancelamento de $\oplus$

Tabela 5.16: Prova de corretude do funcionamento do algoritmo Blowfish

Tendo demonstrado que $L_{16}^d = L_0$ e $R_{16}^d = R_0$, está demonstrado que $L_{16}^d R_{16}^d = L_0 R_0$ e, portanto, que os dados recuperados após a decodificação equivalem aos dados anteriores à codificação.

Para facilitar o acompanhamento da prova de corretude do Blowfish, a tabela a seguir providencia uma lista das equações temporárias definidas.

Número	Equação
Eq1	$L_1^d = P_{17} \oplus R_{15}$
Eq2	$R_1^d = L_{15} \oplus P_{16}$
Eq3	$R_2^d = L_1^d \oplus P_{17}$
Eq4	$R_2^d = R_{15}$
Eq5	$L_2^d = (L_{15} \oplus P_{16}) \oplus F(R_{15})$
Eq6	$R_3^d = F(R_{15}) \oplus L_{15}$
Eq7	$R_{14} = F(R_{15}) \oplus L_{15}$
Eq8	$R_3^d = R_{14}$
Eq9	$L_3^d = P_{15} \oplus \left(L_{14} \oplus F(L_{13} \oplus P_{14})\right)$
Eq10	$R_{13} = L_{14} \oplus F(L_{13} \oplus P_{14})$
Eq11	$L_3^d = R_{13} \oplus P_{15}$
Eq12	$R_{13} = L_3^d \oplus P_{15}$
Eq13	$L_4^d = R_{14} \oplus F(L_3^d \oplus P_{15})$

Tabela 5.17: Listas de equações definidas nos comentários da prova de corretude do algoritmo Blowfish

PUC-Rio - Certificação Digital Nº 1521382/CA

6 Conclusão

Ao fechamento deste trabalho, há diversas observações a serem feitas sobre a sua realização, a primeira das quais diz respeito à sua relevância para a área de Ciência da Computação. Conforme mencionado na seção 1.1, a descrição dos algoritmos de criptografia aqui apresentados constitui um conjunto de provas formais que ainda não fora, do conhecimento do autor, descrito no Lean Theorem Prover.

Em decorrência disso, foram encontradas algumas dificuldades na realização deste trabalho. Por exemplo, para a realização da prova de funcionamento do algoritmo RSA, tiveram de ser definidas diversas propriedades da aritmética modular (descritas na seção 5.2.2) para que estas fossem utilizadas na construção da prova. Espera-se, portanto, que este trabalho contribua para a comunidade científica das seguintes formas:

- Servindo de inspiração para que sejam realizados mais trabalhos relacionados à prova de corretude de algoritmos e protocolos da área de criptografia, sejam estes realizados no Lean Theorem Prover ou em outros assistentes de provas
- Permitindo que trechos de código das provas descritas neste documento sejam reutilizados, parcialmente ou integralmente, por outros trabalhos realizados no Lean Theorem Prover, economizando tempo e esforço para os seus realizadores.

Em segundo lugar, o autor pretende ressaltar que, independentemente da existência de futuros trabalhos realizados por terceiros com inspiração neste, pretende-se dar continuidade a este trabalho:

- Expandindo o conjunto de provas aqui desenvolvidas de modo a incluir mais algoritmos de codificação e decodificação de dados,
- Incluindo neste conjunto de provas protocolos de segurança, e
- Expandindo o trabalho para incluir provas desenvolvidas em outros assistentes de provas.

Por fim, é fundamental ressaltar que os conhecimentos adquiridos na realização deste trabalho, tanto na área de criptografia quanto em relação a prova interativas de teoremas, mostraram-se fundamentais para formação do autor como mestre na área de informática.

PUC-Rio - Certificação Digital Nº 1521382/CA

7 Referências bibliográficas

1. J. Avigad, L. de Moura, S. Kong, F. van Doorn, J. von Raumer. “The Lean Theorem Prover”. Microsoft Research, Carnegie Mellon University.

2. Par, Christof. Pelzl, Jan. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer. 2010.

3. J. von Raumer, J. Avigad, S. Awodey, G. Snelting, F. Herrlich. “Formalization of Non-Abelian Topology for Homotopy Type Theory”.

4. Hoffstein, Jeffrey. Pipher, Jill. Silverman, Joseph. *An Introduction to Mathematical Cryptography: Undergraduate Texts in Mathematics.* Springer. 2010.

5. Ferguson, Niels. Schneider, Bruce. Kohno, Tadayoshi. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing. 2010.

PUC-Rio - Certificação Digital Nº 1521382/CA