

HACSpec Extractor

Raul Mendiratta

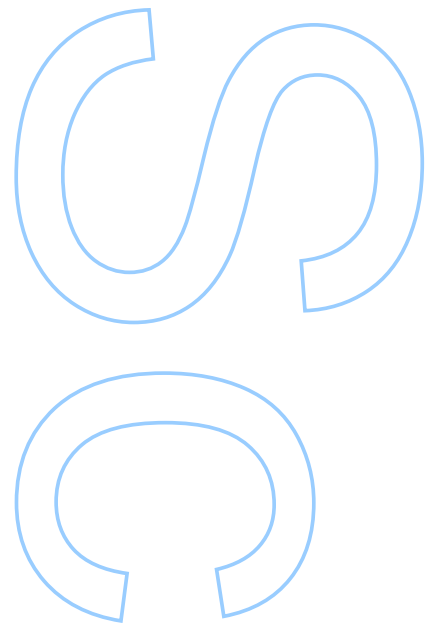
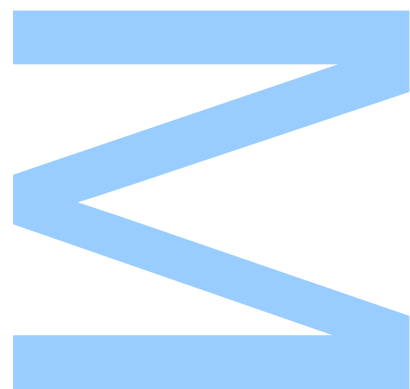
Mestrado Integrado em Engenharia de Redes e Sistemas
Informáticos

[Departamento de Ciências de Computadores](#)

2021

Orientador

[Prof. Manuel Barbosa](#), Faculdade de Ciências

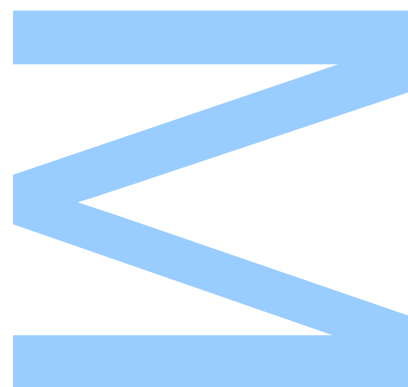




Todas as correções determinadas
pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ____/____/____



UNIVERSIDADE DO PORTO

DISSERTAÇÃO DE MESTRADO

HACSpec Extractor

Autor:

Raul MENDIRATTA

Orientador:

Manuel BARBOSA

Dissertação submetida

para a obtenção do grau de Mestre em Engenharia de Redes e Sistemas Informáticos

na

Faculdade de Ciências da Universidade do Porto

Departamento de Ciências de Computadores

12 de dezembro de 2021

“ Not only is the Universe stranger than we think, it is stranger than we can think. ”

Werner Heisenberg

Agradecimentos

Gostaria de agradecer ao Professor Doutor Manuel Barbosa pela sua disponibilidade, apoio, e constante boa disposição. Foi uma sorte para mim conhecer o Professor que se mostrou uma ajuda essencial nesta fase da minha vida.

A presente investigação fez parte de uma bolsa de iniciação à investigação da FCT no INESC. Um forte agradecimento à equipa do INESC pela disponibilidade e grande competência.

Agradeço aos meus amigos que sempre estiveram ao meu lado. Tanto nos momentos de festa como nos momentos difíceis.

Um grande agradecimento à minha mãe por estar sempre disponível para me ajudar e me acompanhar neste percurso.

Agradeço à Maria por sempre me ajudar e por ser uma forte inspiração de perseverança, resiliência e amor.

Ao meu Pai por ser quem mais me ensinou e aquele que sempre me guiou nas fases mais difíceis da minha vida. Dedico esta dissertação ao meu pai.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto

Departamento de Ciências de Computadores

Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos

HACSpec Extractor

by [Raul MENDIRATTA](#)

Today, the cryptographic software constitutes a crucial component of communication technologies. Servers, web-browsers, wireless technologies, cryptocurrencies; they all employ some kind of cryptographic software. This software must satisfy criteria of efficiency and security for these technologies to function properly and according to expectations. Consequently, proving the security of cryptographic software has been an emerging area of study in the field of computer sciences. To prove the security of a given encryption software, we must prove the security of the algorithm as well as its implementation. HACSpec is a recent collective effort of a group of researchers to create a high-level language to aid in the development of cryptographic algorithms. HACSpec's programs can be run and tested. Easycrypt is a proof assistant aimed at crypto proofs. In this dissertation we present a translator from HACSpec to Easycrypt. This translation can be used as reference proof for a Jasmin implementation. We contribute with the new translator and a translation of the poly1305 encryption algorithm in HACSpec. We validate this translation with an equivalence proof from our translation to a previous existing proof.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto

Departamento de Ciências de Computadores

Mestrado Integrado em Engenharia Física

Titulo da Tese em Português

por [Raul MENDIRATTA](#)

Atualmente, o software criptográfico constitui uma parte crucial das tecnologias de comunicação. Servidores, web-browsers, tecnologias sem fio, criptomoedas, todos estes empregam algum tipo de software criptográfico. Um software criptográfico que não seja seguro e eficiente pode comprometer a segurança e a eficiência destas tecnologias. Provar a segurança de software criptográfico tem sido uma recente área de estudo. Para provar que um software criptográfico é seguro é necessário provar que tanto algoritmo criptográfico como a implementação do algoritmo são seguros. Nesta dissertação é apresentado um tradutor de a linguagem HACSpec para linguagem EasyCrypt. O HACSpec é um projeto que tem como objetivo criar uma linguagem de alto nível para desenvolver algoritmos criptográficos. Os programas em HACSpec podem ser executados e testados O EasyCrypt é um assistente de provas orientado a provas de segurança de criptografia. A especificação em EasyCrypt pode servir como uma prova referência para uma implementação em Jasmin. Contribui-se com um tradutor e uma tradução do poly1305 em HACSpec para EasyCrypt. Valida-se esta tradução como uma prova de equivalência entre uma prova existente e a tradução.

Conteúdo

Acknowledgements	v
Abstract	vii
Resumo	ix
Conteúdo	xi
Lista de Figuras	xv
Glossário	xvii
1 Introdução	1
1.1 Motivação	1
Provar Segurança	1
Implementar Criptografia	2
1.2 Objectivos	2
1.3 Organização	3
2 Enquadramento Tecnológico	5
2.1 ChaCha20-Poly1305	5
2.1.1 ChaCha20	5
<i>ChaCha20 Quarter Round</i>	6
<i>ChaCha Block Function</i>	6
Encriptação	7
2.1.2 Poly1305	8
Algoritmo	8
2.1.3 Implementar ChaCha20-Poly1305	9
2.2 Rust	10
2.3 Jasmin	13
2.3.1 Linguagem	14
Tipos	14
Tipos de Armazenamento	14
Expressões	14
<i>Statements</i>	15
2.3.2 Compilador	15

3	EasyCrypt	17
3.1	Assistentes de Provas	17
3.2	EasyCrypt	18
3.2.1	Sintaxe	18
	Declaração de Tipos	19
	Funções Anónimas	19
	Operadores	19
3.3	Jasmin-EasyCrypt	21
4	HACSpec	25
4.1	HACSpec	25
4.2	Linguagem	28
4.3	Biblioteca	28
4.4	HACSpec-Easycrypt	30
5	Desenvolvimento	33
5.1	Concepção	33
5.1.1	Arquitetura do Tradutor	33
5.1.2	Alterações necessárias	35
	<i>Imports</i>	35
	Declarações	35
	<i>snake_case</i>	35
	Tipos e Nomes de Função	36
	Corpos Finitos	36
	Funcionamento do Tradutor	37
5.2	Implementação	38
	<i>Imports</i>	38
	Declarações e Funcionamento do Tradutor	38
	<i>snake_case</i>	39
	Tipos e nomes de função	39
	Corpos Finitos	40
6	Resultados e Validação	41
	Metodologia de Validação	41
6.1	Resultados	41
	Declaração e Funcionamento do Tradutor	42
	Tipos e nomes de Funções	43
	Corpos Finitos	45
	Discussão	45
7	Conclusão	47
	Prospectiva	47
	Trabalho Realizado	47
	Trabalho Futuro	48
	Bibliography	48

A	Definição da linguagem HACSpec	53
B	Traduções geradas.	55

Lista de Figuras

1.1	Diagrama explicativo do HACSpec.	3
2.1	Parte da implementação do chacha20 em Rust.	11
2.2	Exemplo de código em Jasmin. [1]	15
3.1	Processo de implementação/verificação de cifra em Jasmin e EasyCrypt. . .	22
3.2	Especificação do poly1305 em EasyCrypt [1]	23
4.1	Pseudocódigo da função Poly, do poly1305rfc7539.	26
4.2	Implementação da função poly em HACSpec.	27
4.3	Definição de um inteiro num corpo finito em HACSpec.	30
4.4	Processo de implementação/verificação de cifra em Jasmin e EasyCrypt com HACSpec.	30
5.1	Mensagem de erro do compilador de HACSpec.	34
5.2	Arquitetura do Tradutor.	34
5.3	Parte de código do Poly1305 gerado pelo tradutor.	35
5.4	Parte de código do Poly1305 gerado pelo tradutor.	35
5.5	Função <code>le_bytes_to_num</code> em HACSpec, à esquerda, e em EasyCrypt à direita	36
5.6	Declaração de corpos finitos, à esquerda em HACSpec, e em EasyCrypt à direita.	37
5.7	<i>Workflow</i> do tradutor	37
5.8	Definição de <code>array_16_from_seq</code> in EasyCrypt	38
5.9	<i>Workflow</i> do tradutor.	38
5.10	Parte do código do tradutor.	39
5.11	Parte do código do tradutor.	39
5.12	Parte do código do tradutor.	40
5.13	Tradução correta da declaração de corpos finitos.	40
6.1	Processo Validação	41
6.2	Tradução de declaração de tipos.	42
6.3	Tradução de declaração de tipos.	43
6.4	Tradução de invocações de funções.	44
6.5	Declaração de operadores.	44
6.6	Tradução de declaração de tipos.	45

Glossário

AST	Abstract Syntax Tree
EC	EasyCrypt

Capítulo 1

Introdução

Neste capítulo, pretende-se contextualizar a temática da dissertação. Está dividido em dois sub-capítulos. No primeiro, compreende-se a motivação deste projeto. De seguida, estima-se as contribuições produzidas pelo trabalho desenvolvido.

1.1 Motivação

Actualmente, o software criptográfico é um componente crucial nas tecnologias de comunicação. A generalidade das comunicações aplica algum tipo de software criptográfico. Os requisitos para um software criptográfico variam consoante a tecnologia, no entanto existem dois que são sempre necessários. O software tem de ser seguro e eficiente. Se um software criptográfico não for seguro pode comprometer a privacidade da comunicação, ou a integridade das mensagens. Se um software não for eficiente, pode não ser viável. Suponha que a encriptação de as respostas de um servidor demoram 0.1 segundos, então este servidor demoraria 50 segundos para responder a 500 pedidos. Garantir estes dois requisitos não é trivial.

Provar Segurança Provar que um software criptográfico é seguro é uma área de estudo da criptografia. Criptógrafos e Matemáticos recorrem a provas formais para provar que a cifra é segura. Existem vários métodos de provar que uma cifra é segura. Frequentemente, primeiro definem o que é a segurança do sistema e depois recorrem a uma redução para uma prova de que um determinado problema matemático seja irrealista resolver por um atacante.

Existem duas provas diferentes. Existe a prova de segurança ao algoritmo criptográfico e a prova de que a implementação é correta e segura. Ambas são necessárias pois pode haver ataques ao algoritmo ou à implementação. Ataques ao algoritmo podem usar técnicas como a análise estatística. Ataques à implementação geralmente são ataques que tentam obter informação sobre a execução da implementação da cifra. Em geral, as provas ao algoritmo recorrem ao método das reduções a um problema matemático. As provas à implementação são diferentes.

A prova à implementação é constituída por duas provas. A prova de que a implementação executa de acordo com a especificação do algoritmo e a prova de que a implementação é segura. Isto conjuga-se com outra área de estudo que é a verificação formal de software. Verifica-se formalmente se a implementação está correta de acordo com a especificação do algoritmo criptográfico.

As provas de segurança são processos complicados e que habitualmente necessitam de especialistas de diferentes áreas como a de criptografia e a de verificação formal de software.

Implementar Criptografia Para alcançar uma implementação otimizada de uma cifra é necessário uma linguagem de baixo nível tipo *assembly*. Programar nestas linguagens não é do conhecimento geral dos programadores. Além disso, as especificações das cifras normalmente vem com um pseudocódigo descritivo e/ou uma implementação numa linguagem de alto nível, como C. Isto cria uma lacuna entre o algoritmo criptográfico e a implementação. Fechar esta lacuna tem sido uma área de estudo recente no âmbito da criptografia.

1.2 Objectivos

O HACSpec é um projeto que pretende criar uma linguagem de alto nível para desenvolver software criptográfico. Os programas em HACSpec podem ser executados, testados e traduzidos para FStar. O EasyCrypt é uma plataforma para assistir nas provas de segurança. O Jasmin é uma linguagem para desenvolver implementações eficientes e seguras. O HACSpec contém um tradutor da linguagem HACSpec para a linguagem EasyCrypt. Este tradutor não produz uma tradução correta. Esta dissertação tem como objectivo a alteração deste tradutor para produzir uma tradução correta da linguagem HACSpec para EasyCrypt.

Traduzindo os programas de HACSpec para EasyCrypt estes podem ser verificados e provados. Adicionalmente a tradução gerada em EasyCrypt pode servir como prova de referência para uma implementação em Jasmin. Com este tradutor estabelece-se uma ponte entre o algoritmo criptográfico e a implementação. Esta dissertação traz como contribuição um tradutor capaz de traduzir o algoritmo criptográfico poly1305 e parcialmente o chacha20.

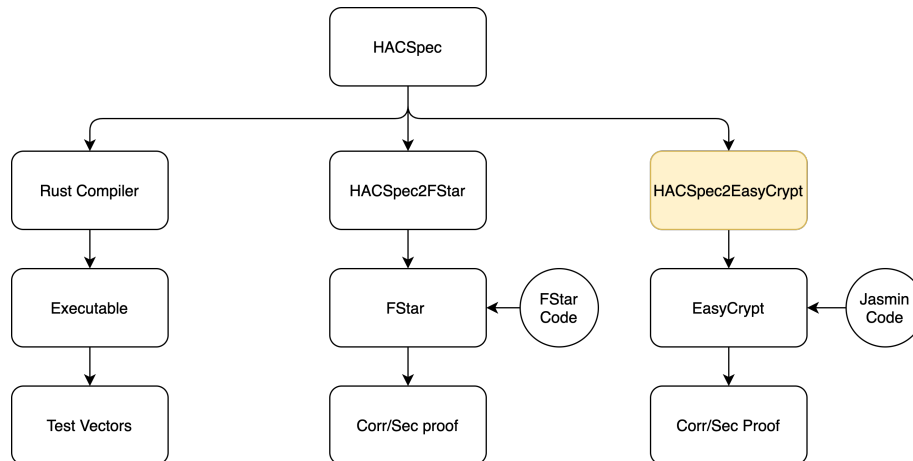


FIGURA 1.1: Diagrama explicativo do HACSpec.

1.3 Organização

Esta dissertação está dividida em 7 capítulos.

No capítulo 1 faz-se uma introdução à dissertação. Apresenta-se a motivação da dissertação e estima-se as contribuições produzidas.

No capítulo 2 faz-se um enquadramento tecnológico com uma apresentação do sistema de encriptação chacha20-poly1305 e uma descrição de algumas características da linguagem Rust e do Jasmin.

No capítulo 3 descreve-se a linguagem EasyCrypt. Para contextualizar, expõe-se os assistentes de prova e descreve-se o EasyCrypt e a sua ligação com o Jasmin.

No capítulo 4 entende-se o HACSpec e descreve-se a sua linguagem e biblioteca. Adicionalmente, conecta-se o HACSpec com o EasyCrypt.

No capítulo 5 descreve-se o processo de desenvolvimento do tradutor. É detalhada a concepção do tradutor com a apresentação do estado do tradutor e as alterações necessárias.

No capítulo 6 apresenta-se as traduções produzidas a serem validadas pelo verificador do EasyCrypt. Adicionalmente, é detalhada a metodologia para a validação.

No capítulo 7 expõe-se as conclusões. Apresenta-se uma perspectiva do âmbito tecnológico e refere-se o trabalho realizado. No final, indica-se o trabalho futuro.

Capítulo 2

Enquadramento Tecnológico

Neste capítulo, começa-se por descrever o sistema de encriptação autenticada ChaCha20-Poly1305 como um exemplo demonstrativo dos desafios da implementação da criptografia. De seguida, na secção 2.2, apresenta-se o Rust como uma opção de linguagem inicial para a implementação de especificações de criptografia. Além disso, a implementação do tradutor foi desenvolvida na linguagem Rust. Finalmente, na secção 2.3, introduz-se o Jasmin como uma plataforma para desenvolver e implementar criptografia segura e eficiente.

2.1 ChaCha20-Poly1305

O AEAD-ChaCha20-Poly1305 é um sistema de encriptação autenticada construído por D. J. Bernstein para ser suportado pelo TLS 1.3. O sistema de encriptação aceita uma chave de 256 *bits*, um *nonce* de 96 *bits* (diferente para cada computação com a mesma chave) e um texto de qualquer tamanho para produzir um texto encriptado com o mesmo tamanho que do texto original, e um *tag* de 128 bits para autenticação. Nas seguintes secções são apresentados em detalhe o ChaCha20 e o Poly1305.

2.1.1 ChaCha20

O ChaCha20 é uma cifra de fluxo que aceita uma chave de 256 *bits*, um *nonce* de 96 bits e um texto de qualquer tamanho. O texto é encriptado ao ser *XORed* (soma módulo 2) com uma chave de fluxo de tamanho igual. Esta chave de fluxo é gerada ao concatenar os blocos gerados pelas invocações sucessivas à função *ChaCha20 Block function*, que recebe como argumento a chave de 256 bits e *nonce* e um contador do bloco. A desencriptação

é realizada com o mesmo procedimento, gerando a mesma a chave de fluxo para executar a operação *XOR* com o texto encriptado.

ChaCha20 Quarter Round A operação básica do ChaCha20 é a sua operação *quarter round*. Esta operação é realizada em quatro inteiros sem sinal de 32 *bits*, identificados por *a*, *b*, *c* e *d*. A operação é a seguinte:

```

1  1.  a = a + b; d = d XOR a; d shift-left 16;
2  2.  c = c + d; b = b XOR c; b shift-left 12;
3  3.  a = a + b; d = d XOR a; d shift-left 8;
4  4.  c = c + d; b = b XOR c; b shift-left 7;
```

Note que as adições são adições modulo 32.

ChaCha Block Function Os inputs do *ChaCha Block* são:

- uma chave de 256 *bits*, tratada como uma concatenação de 8 inteiros de 32 *bits little-endian*.
- um *nonce* de 96 *bits*, tratado como uma concatenação de 3 inteiros de 32 *bits little-endian*
- um contador de 32 *bits*, tratado como um inteiro de 32 *bits little-endian*.

Estes argumentos formam o estado inicial do ChaCha20 da seguinte maneira:

```

1      c c c c k k k k k k k k b n n n
2
3      c=constante k=chave b=contador do bloco n=nonce
```

Onde nas posições 0-3 são as seguintes constantes(*c*): 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574, nas posições 4-11 está a chave, na posição 12 está o contador e nas posições 13-15 está o *nonce*.

É aplicado 8*10 *quarter rounds*(ou 20 *rounds*) ao estado da seguinte maneira:

```

1      Quarter_round(state, 0, 4, 8,12)
2      Quarter_round(state, 1, 5, 9,13)
3      Quarter_round(state, 2, 6,10,14)
4      Quarter_round(state, 3, 7,11,15)
```

```
5     Quarter_round(state, 0, 5,10,15)
6     Quarter_round(state, 1, 6,11,12)
7     Quarter_round(state, 2, 7, 8,13)
8     Quarter_round(state, 3, 4, 9,14)
```

Este estado é somado com o anterior e retorna-se o estado serializado. O pseudocódigo da função é como se indica:

```
1 chacha20_block(key, counter, nonce):
2     state = constants | key | counter | nonce
3     working_state = state
4     for i=1 upto 10
5         quarter_rounds(working_state)
6     end
7     state += working_state
8     return serialize(state)
9 end
```

Encriptação O bloco gerado pela função *chacha20_block* é depois *XORed* concatenado com blocos anteriores para formar a chave de fluxo. Alternativamente, é possível ir gerando os blocos e *XORing* com o texto. Pode-se verificar o seguinte pseudocódigo do ChaCha20:

```
1 chacha20_encrypt(key, counter, nonce, plaintext):
2     for j = 0 upto floor(len(plaintext)/64)-1
3         key_stream = chacha20_block(key, counter+j, nonce)
4         block = plaintext[(j*64)..(j*64+63)]
5         encrypted_message += block ^ key_stream
6     end
7     if ((len(plaintext) % 64) != 0)
8         j = floor(len(plaintext)/64)
9         key_stream = chacha20_block(key, counter+j, nonce)
10        block = plaintext[(j*64)..len(plaintext)-1]
11        encrypted_message += (block^key_stream)[0..len(plaintext)%64]
12    end
```

```

13     return encrypted_message
14     end

```

2.1.2 Poly1305

O Poly1305 produz uma *tag* de 16 *bytes* de uma mensagem de tamanho variável, usando uma chave secreta k . Esta *tag* é anexada à mensagem. O receptor deverá autonomamente derivar a mesma chave k para depois calcular a *tag*. Se a *tag* recebida for igual à *tag* calculada, o receptor pode assegurar-se que só o remetente poderia ter enviado a mensagem.

Algoritmo O poly1305 recebe uma chave utilizada só uma vez de 32 *bytes* e produz um *tag* de 16 *bytes*. A chave é dividida em duas partes, r e s , em que são as duas tratados como números de 16 octetos *little endian*. Antes de r ser usado, é primeiro modificado da seguinte maneira: $r[3], r[7], r[11]$, e $r[15]$ devem ter os top quatro bits "limpos" (ser menor que 16) e $r[4], r[8]$, e $r[12]$ devem ter os dois *bottom* bits "limpos" (ser divisível por 4). Na presente dissertação é assumido que a chave é gerada como um numero pseudo-aleatorio de 256 bits. O *tag* é gerado através dos seguintes passos. A mensagem m é dividida em blocos de 16 *bytes* m_i em que $i \in [1, 2, \dots]$. Cada bloco m_i é lido como um valor de 16 *bytes little endian* e depois cada centésimo de vigésimo nono bit é posto a um (o último bloco é tratado de maneira diferente). A *tag* t é calculada ao acumular sequencialmente cada número(m_i) a um estado inicial $a_0 = 0$ de acordo com a seguinte formula: $a_i = (a_{i-1} + b_i) * r \pmod{p}$ para cada $i \in [1, 2, \dots]$ e onde p é um numero primo com o valor $2^{130} - 5$. Finalmente a chave secreta é somada ao acumulador e *tag* t é simplesmente os 128 *bits* mais baixos serializados na ordem *little endian* [1]. A escolha do numero primo $2^{130} - 5$ facilita as operações de divisão em software bem como em hardware [2]. Descreve-se o seguinte pseudocódigo do poly1305:

```

1  clamp(r): r &= 0xffffffffc0xffffffffc0xffffffffc0xffffffff
2      poly1305_mac(msg, key):
3          r = (le_bytes_to_num(key[0..15]))
4          clamp(r)
5          s = le_num(key[16..31])
6          accumulator = 0
7          p = (1<<130)-5

```

```

8      for i=1 upto ceil(msg length in bytes / 16)
9          n = le_bytes_to_num(msg[((i-1)*16)..(i*16)] | [0x01])
10         a += n
11         a = (r * a) % p
12     end
13     a += s
14     return num_to_16_le_bytes(a)
15 end

```

2.1.3 Implementar ChaCha20-Poly1305

Apesar da descrição criar a percepção de que a sua implementação é simples, a implementação com base nas descrições e no pseudocódigo pode-a tornar num processo complexo. Por exemplo, na especificação do poly1305 existe a seguinte instrução: $a_i = (a_{i-1} + b_i) * r \pmod{p}$. Repara-se que a_i é um inteiro de 16 *bytes* (128 bits) módulo $2^{130} - 5$. É necessário dividir este inteiro em inteiros de 64 *bits* ou de 32 bits para as operações de soma e de módulo. Para implementar estas operações é necessário um bom conhecimento de aritmética modular. Por exemplo, a instrução $r * a$ do pseudocódigo demonstrado anteriormente resulta na seguinte sequência de instruções (em Rust):

```

1  /* a = r * a */
2  ...
3  let mut d0: u128 = a0 as u128 * r0 as u128;
4  let mut d:  u128 = a1 as u128 * s2 as u128;
5  d0 += d;
6  d = a2 as u128 * s1 as u128;
7  d0 += d;
8
9  d1 = a0 as u128 * r1 as u128;
10 d = a1 as u128 * r0 as u128;
11 d1 += d;
12 d = a2 as u128 * s2 as u128;
13 d1 += d;
14 d2 = a0 as u128 * r2 as u128;

```

```
15 d = a1 as u128 * r1 as u128;  
16 d2 += d;  
17 d = a2 as u128 * r0 as u128;  
18 d2 += d;  
19 ...
```

Ainda que, se consiga chegar a uma compreensão do algoritmo e a uma implementação, que até passa em todos os vetores de teste, é possível que esta possa não ter utilidade. A execução da implementação pode não ser suficientemente rápida para as exigências atuais dos servidores. Frequentemente, para melhorar a eficiência da implementação será preciso recorrer a instruções de "baixo nível" como o de *assembly*, de maneira que se construa uma implementação eficiente. Isto exige um grande nível de conhecimento de linguagens de baixo nível que não é comum na generalidade dos programadores. A implementação pode estar sujeita a *side-channel attacks* e erros matemáticos. Estes erros podem passar despercebidos durante a testagem da implementação. Adicionalmente, um atacante pode tirar partido destes erros. Certificar que não existem erros com a verificação da implementação é demorado e requer um alto nível de perícia, tornando-se impraticável.

2.2 Rust

A linguagem Rust começou por ser um projeto de Graydon Hoare no ano de 2006, e teve o seu lançamento oficial em 2015 [3]. A linguagem foi desenvolvida com o objetivo de ser uma linguagem que produza programas seguros e eficientes.

Existem duas abordagens em que uma linguagem pode aumentar segurança: pode tornar fácil para o programador escrever código seguro ou pode tornar difícil para o programador escrever código inseguro. O Rust certamente usa a última abordagem.

A sintaxe do Rust é semelhante à de C e C++*. Na figura 2.1 pode-se verificar um pequeno exemplo de uma função em Rust.

As principais características da linguagem Rust que a destingue de outras linguagens são:

- **variáveis imutáveis**

*Optou-se por não fazer uma síntese exaustiva da sintaxe do Rust, apesar de se encontrar pertinência a referenciar

```
1 //the chacha20 block function
2 fn chacha20_block(key: &String, counter: u32, nonce: &String) -> Vec<u8>{
3
4     let mut state: [u32;16] = [0;16];
5     build_state(&mut state,key,counter,nonce);
6     let working_state = state;
7
8     for _i in 0..10 {
9         inner_block(&mut state);
10    }
11
12    for i in 0..16 {
13
14        state[i] = state[i].wrapping_add(working_state[i]);
15    }
16
17    let serialized_state = serialize(&mut state);
18    serialized_state
19 }
20
21
```

FIGURA 2.1: Parte da implementação do chacha20 em Rust.

As variáveis por predefinição são imutáveis. É necessário indicar que uma variável é mutável para se tornar mutável. Note-se na figura 2.1 a variável mutável *state* e a imutável *working_state*. Isto encoraja o programador a pensar antes de deixar alguma variável ser mutável quando não deveria ser, prevenindo *bugs* imprevisíveis no código.

- **donos**

Todas as variáveis têm um único "dono". O valor apontado pelas variáveis pode ser passado através de referência imutável *&T* ou mutável *& mut T*, as referências mutáveis só podem ser atribuídas a uma só variável numa secção do código. Assim, é possível garantir que não existam várias referências mutáveis para o mesmo valor ao mesmo tempo durante a execução. Por exemplo, o conjunto seguinte de instruções não é válido:

```
1     let mut s = String::from("hello");
2
3     let r1 = &mut s;
4     let r2 = &mut s;
```

```
5
6     println!("{}", r1, r2);
7
```

- **alocação de memória**

O Rust tem um comportamento bem definido sobre a instrução da atribuição (=); cria uma distinção entre *shallow copy*s e *deep copy*s. Por exemplo, o conjunto seguinte de instruções não é válido [4]:

```
1     let s1 = String::from("hello");
2     let s2 = s1;
3
4     println!("{}", s1);
5
```

Isto acontece porque uma atribuição no Rust é de facto aquilo a que os responsáveis pelo desenvolvimento de Rust chamam de *move*. O apontador para a *String* que está na variável *s1* passa para a variável *s2*. A variável *s1* deixa de ser referenciável. Para copiar o valor da *string* é necessário usar o método *clone()* que faz um *deep copy* da variável.

- **tempos de vida**

O Rust permite a indicação de até que parte do código é que a variável deixa de ser usada para, assim, não permitir expressões com apontadores para o valor dessa variável para além dessa parte do código. Por exemplo, a seguinte função não é válida:

```
1     fn longest(x: &str, y: &str) -> &str {
2         if x.len() > y.len() {
3             x
4         } else {
5             y
6         }
7     }
8
```

Isto acontece porque não se consegue saber que variável vai ser retornada, x ou y logo não é possível saber que valor é que $\&str$ vai apontar, o que pode causar uma referência pendente. O Rust resolve isto ao obrigar o programador a anotar o tempo de vida. Por exemplo, a seguinte instrução é válida:

```
1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
2     if x.len() > y.len() {  
3         x  
4     } else {  
5         y  
6     }  
7 }  
8
```

Note-se que foi anotado o tempo de vida com $'a$. Isto significa que as variáveis como parâmetros da função e a variável de retorno têm de ter o mesmo tempo de vida. Assim, o compilador do Rust consegue calcular quanto tempo precisa de manter aquela referência.

O Rust foi a linguagem de escolhida para o desenvolvimento do HACSpec. A razão desta escolha reside no fato de o Rust ter propriedades interessantes, como já anteriormente mencionadas, e também de existir a expectativa que o Rust seja uma linguagem amplamente usada no futuro.

2.3 Jasmin

Como já foi referido na secção 2.1.3, a implementação de algoritmos criptográficos seguros e eficientes é um processo muito complexo. A consideração da eficiência torna inviável usar uma linguagem de alto nível para implementar algoritmos criptográficos. O código tem de estar otimizado a um nível que é inatingível pelos compiladores de linguagens de alto nível. Adicionalmente, estes compiladores podem introduzir defeitos de segurança. Um exemplo de uma linguagem usada para desenvolver algoritmos criptográficos é a linguagem qhasm [5]. As implementações em qhasm são altamente eficientes e a linguagem tem um nível de abstracção acima de *assembly*. Apesar destas vantagens, as implementações em qhasm não são facilmente acessíveis a uma verificação formal [6]. O ideal seria

uma linguagem com suficiente nível de abstracção, com a finalidade de ser compreensível para o criptografo comum, também com a finalidade de ser formalmente verificada e com suficiente nível de instruções de baixo nível para assim atingir uma implementação otimizada. Com o objetivo de abordar e procurar resolver esta problemática, foi iniciado o projeto Jasmin.

O Jasmin é um projeto colaborativo entre investigadores do INESC TEC, IMDEA, Inria, e a Ecole Polytechnique que tem como objetivo de criar uma linguagem para o desenvolvimento de cifras criptograficas seguras e optimizadas. As implementações em Jasmin têm o objetivo serem eficientes, seguras e corretas [7]. O Jasmin conjuga abstrações de alto nível, como arrays, variáveis e controlo de fluxo, com instruções de baixo nível como instruções de *assembly* e manipulação de *flags* [1].

2.3.1 Linguagem

Nesta secção descreve-se sucintamente a linguagem Jasmin.

Tipos O Jasmin contém um sistema simples de tipos, composta por tipos de *ui* sendo *i* o número de bits do valor que está contido no intervalo {8,32,64,128,256}, o tipo *bool* que representa booleanos e *int* para inteiros sem definição de tamanho. Para além destes tipos, o Jasmin fornece arrays para qualquer dos tipos mencionados anteriormente. Repare-se na figura 2.2, na segunda linha, a declaração de um array do tipo *u256*.

Tipos de Armazenamento Os tipos de armazenamento são usados pelo compilador para definir como os valores estão guardados na memória. Os tipos de memória são: guardar numa *stack*, guardar num registo *reg*, resolvidos na altura da compilação *inline* [6]. A declaração dos tipos de armazenamento é realizada antes da declaração do tipo, como se pode observar na figura 2.2.

Expressões As expressões são construídas através variáveis, constantes, operadores, acessos de array, acessos de memória, *cast* de inteiros para *u64*, condicionais e operações de computação em paralelo [6]. Os operadores incluem constantes de inteiros, operadores aritméticos e de comparação (+, -, *, /, <, >, ≤, ≥, =), operadores lógicos (∧, ∨, ¬) e operadores de *assembly*: soma com *carry*, incrementação/decrementação, *shifts* etc. Acessos de memória são da forma $[x + e]$ em que *x* é a variável do tipo *u64* que representa o

apontador e e é o *offset*. Acessos ao array são da forma $x[e]$ onde x é o identificador do array e e é a expressão do tipo $u64$. Jasmin também permite operações em paralelo(SIMD*) para melhorar a eficiência. Por exemplo, a instrução $z[1] + 4u64 = t$ da linha 9 da figura 2.2, em que z e t são do tipo $u256$, executa quatro somas em paralelo de $u64$ nas variáveis z e t e guarda o valor na variável $z[1]$.

Statements Os *Statements* do Jasmin são constituídos por atribuições, invocações de função, condicionais, ciclos *for* e *while*. Os *Statements* são sempre finalizados por ;.

```

1 fn carry_reduce(reg u256[5] x, reg u256 mask26) -> reg u256[5] {
2   reg u256[2] z;
3   reg u256 t;
4   z[0] = x[0] >>4u64 26; z[1] = x[3] >>4u64 26;
5   x[0] &= mask26; x[3] &= mask26;
6   x[1] +4u64= z[0]; x[4] +4u64= z[1];
7   z[0] = x[1] >>4u64 26; z[1] = x[4] >>4u64 26;
8   t = z[1] <<4u64 2;
9   z[1] +4u64= t;
10  x[1] &= mask26; x[4] &= mask26;
11  x[2] +4u64= z[0]; x[0] +4u64= z[1];
12  z[0] = x[2] >>4u64 26; z[1] = x[0] >>4u64 26;
13  x[2] &= mask26; x[0] &= mask26;
14  x[3] +4u64= z[0]; x[1] +4u64= z[1];
15  z[0] = x[3] >>4u64 26;
16  x[3] &= mask26;
17  x[4] +4u64= z[0];
18  return
19

```

FIGURA 2.2: Exemplo de código em Jasmin. [1]

2.3.2 Compilador

O compilador de Jasmin é formalmente verificado [6]. Isto significa, que é certificado que o compilador preserva a semântica. Logo, para provar que uma implementação em Jasmin está correta, basta provar que o código em Jasmin está correto.

*SIMD(Single Instruction Multiple Data) é uma instrução de máquina que opera sobre vários dados ao mesmo tempo(computação paralela).

Capítulo 3

EasyCrypt

Neste capítulo é apresentado o EasyCrypt. Para contextualizar o EasyCrypt na secção 3.1 é introduzido o conceito de um assistente de prova. De seguida, na secção 3.2 é descrita a linguagem EasyCrypt. Finalmente, na secção 3.3, é descrito o processo de ligação do Jasmin ao EasyCrypt.

3.1 Assistentes de Provas

Uma prova formal é uma prova em que todos os passos lógicos estão explicitados. É um cálculo que segue regras precisas. A notação formal é rigorosa, inequívoca e pode ser verificada mecanicamente. É precisamente na parte "mecânica" que os assistentes de provas atuam [8]. A função de um assistente de prova é verificar que todos estes passos lógicos estão correctos. Os assistentes de provas tem vindo a adquirir notoriedade na área da ciência da computação pois provar um determinado comportamento de um software pode ser um processo complicado e com margem de erro. Os assistentes de provas tem vindo a ajudar a resolver este problema [9] verificando que todos os passos e todas as declarações de um programa estão correctos. Existem vários assistentes de provas conhecidos como o Coq [10], Agda [11], Lean [12] e PVS [13]. Estes assistentes de provas têm como objectivo principal assistir em formalizações de provas de matemática e verificação de software. Apesar de estes assistentes de provas serem fortes nas suas áreas, a sua *framework* é limitada. Existem alguns especialmente orientados para área da prova de segurança e na criptografia [8]. O F* [14] e o EasyCrypt [15] são exemplos destes assistentes de provas e na presente dissertação optou-se descrever em mais detalhe o EasyCrypt.

3.2 EasyCrypt

O EasyCrypt é um assistente de prova orientado à criptografia. Usa o método de provas através de jogos e a lógica de Hoare [16]. Foi utilizado para provar esquemas de encriptação como o CMAC, SHA-3 e o ChaCha20-Poly1305 [8]. Em EasyCrypt os jogos são módulos que consistem em procedimentos e variáveis globais. As variáveis globais são visíveis fora dos módulos e definem o seu estado interno a qualquer altura. Um procedimento é um conjunto de instruções e declarações. Uma instrução pode ser uma declaração, uma amostra aleatória, uma chamada a um procedimento, um ciclo *while* ou uma declaração de retorno. O EasyCrypt permite 3 tipos de lógicas de programa: a lógica de Hoare, a lógica probabilística de Hoare e a lógica relacional probabilística de Hoare [15].

No contexto presente é necessário descrever apenas alguns aspectos da linguagem EasyCrypt como algumas declarações e alguns tipos de dados. Isto acontece porque a tradução do HACSpec para EasyCrypt só usa algumas partes da linguagem EasyCrypt.

3.2.1 Sintaxe

Nesta secção é descrita em pormenores alguma parte da sintaxe do EasyCrypt. É apenas descrita a parte da sintaxe relevante ao projecto. A descrição de toda a linguagem pode ser consultada na referência [17].

Um *script* em EasyCrypt consiste num conjunto de passos separados por (.). Os passos podem ser [17]:

- declaração de **Tipos**;
- declaração de **Operadores** e Predicados;
- declaração de módulos ou de tipos de módulos;
- axiomas de estado ou lemas;
- aplicar táticas;
- requerer teorias;
- entrar ou sair de teorias ou secções;
- imprimir tipos, operadores, predicados, modulos, tipos de módulos, axiomas e lemas;
- procurar por lemas que envolvem operadores;

Declaração de Tipos No código abaixo pode-se observar uma declaração do tipo t em EasyCrypt. Os tipos contidos no EasyCrypt são inteiros(int), booleanos(bool), reais(real), e o tipo *singleton* (unit). Na segunda linha do código também se pode observar um outro tipo de declaração de tipo. Aqui, se receber um valor v que seja *int*, então $x : \text{int}$ e $f = \text{int} \implies \text{int}$ [17].

```

1 type t = { x : int; y : bool; z: bool; k: real; i: unit }.
2 type 'a t = { x : 'a; f : 'a -> 'a: }.
3

```

Para além destes tipos de dados, o EasyCrypt também fornece *words*, que são um conjunto de bits com operações *bitwise* e *array* que são conjuntos de *words*. Esta parte ainda está em desenvolvimento mas é relevante mencionar para melhor compreender a secção 3.3.

Funções Anónimas No código abaixo é indicado a declaração de uma função anónima em EasyCrypt, onde x é identificador, $t1$ é tipo e e é expressão. Se e for do tipo $t2$ então a função vai ser do tipo $t1 \implies t2$. Na segunda linha do código a função vai ser avaliada com o valor 1. O EasyCrypt também contém expressões *let* como se pode observar na quarta linha do código. Esta expressão é equivalente à da terceira linha [17].

```

1 fun (x : t1) => e,
2 (fun x y z => y) 0 1 false
3 (fun x : t => e)e
4 let x : t = e in e
5

```

Operadores Os operadores são declarados especificando o tipo e a expressão a ser avaliada. No código abaixo pode se ver exemplos de operadores. Repare-se que y e z vão ser avaliados com o valor 1, porque f é uma função *curried*[17].

```

1 op x : int = 3.
2 op f : int -> bool -> int = fun (x : int) (y : bool) => x.
3 op g : bool -> int = f 1.
4 op y : int = g true.
5 op z = f 1 true.
6

```

A declaração de f pode ser abreviada para:

```
1 op f (x : int) (y : bool) = x.
2
```

Operadores polimórficos podem ser declarados da seguinte maneira:

```
1 op g ['a, 'b] (x : 'a, y : 'b) = x.
2 op f (x : 'a, y : 'b) = x.
3 op a = f true 0.
4 op b = f 0 false.
5
```

Aqui repare-se que a função f pode receber dois tipos diferentes, criando assim a avaliação do operador a como *true* e o operador b como 0.

Também se pode declarar operadores abstractos:

```
1 op x : int.
2 op f : int -> int.
3 op g ['a, 'b] : 'a -> 'b -> 'a.
4
```

O EasyCrypt também permite a declaração de operadores binários. Estes são declarados com parêntesis e se forem usados como prefixos também será necessário o uso de parêntesis. Se forem usados como operador *infix*, não precisam dos parêntesis.

```
1 op (--) ['a, 'b] (x : 'a) (y : 'b) = x.
2 op x : int = (--) 0 true.
3 op y : bool = true -- 0.
4
```

Também é possível declarar operadores *mixfix*. Estes tem de ser declarados com aspas e se forem usados como prefixo também será necessários usar aspas. Se forem usados como operador *mixfix* não precisam das aspas. É de notar que o operador `[]` tem maior precedência de que uma função sendo que não se pode omitir os parêntesis caso o objetivo seja executar primeiro a função e depois o operador `[]`, como se pode ver no operador `l` e `z`.

```

1 op "[]" : int = 3.
2 op x : int = [].
3 op "_.[_]" : int -> int -> bool.
4 op y : bool = "_.[_]" 3 4.
5 op y' : bool = 3.[4].
6 op "_.[_<-_]" : int -> int -> int -> bool.
7 op g : bool = 3.[4 <- 5].
8 op f : int -> int.
9 op l : bool = (f 3).[4].
10 op z : bool = (f 3).[4 <- 5].
11
```

O EasyCrypt já vem com alguns operadores predefinidos:

```

1 op (=) ['a]: 'a -> 'a -> bool.
2 op [!] : bool -> bool.
3 op (||) : bool -> bool -> bool.
4 op (\/) : bool -> bool -> bool.
5 op (&&) : bool -> bool -> bool.
6 op (/\\) : bool -> bool -> bool.
7 op (=>) : bool -> bool -> bool.
8 op (<=>) : bool -> bool -> bool.
9
```

Assim é possível compreender que o EasyCrypt contém características de uma linguagem de programação funcional com definição de tipos e de funções o uso de funções *lambda* e com *currying*.

3.3 Jasmin-EasyCrypt

O Jasmin pode ser utilizado com o EasyCrypt para provar que a implementação em Jasmin está correta e segura. Para atingir a prova, segue-se a seguinte abordagem [18]:

- Transpõe-se a especificação técnica para o EasyCrypt. Note-se que esta **tem de ser realizada manualmente**. Consequentemente, a própria **transposição tem de ser**

verificada. O processo de transposição é complexo. É necessário entrar em pormenores relacionados com os tipos de dados da especificação técnica. Este processo é extenso e pode levar a erros. Além disso, esta tradução tem de ser o mais exacto possível com a especificação. Alcançada uma transposição correta, prova-se que a especificação é segura e correta no sentido criptográfico. Note-se que esta prova é relacionada com o algoritmo criptográfico e não com a implementação.

- Traduz-se a implementação em Jasmin para EasyCrypt. Utiliza-se esta tradução para provar que a implementação está funcionalmente correta de acordo com a especificação. Isto é possível com a implementação da semântica do Jasmin no EasyCrypt e com um mapeamento dos tipos dados da especificação para a implementação. Repare-se que o mapeamento depende do nível de detalhe da especificação. A especificação pode conter informação detalhada ao nível dos *bytes* ou pode só ter informação ao nível mais abstracto. Um exemplo disto já foi descrito na secção 2.1.3 em que a especificação do poly1305 não é clara sobre como operar com inteiros de 128 *bits*.
- Assumindo que a especificação em EasyCrypt está correta, a prova de correção garante que o código implementa a especificação técnica.

A figura 3.1 demonstra o processo de construção da prova da implementação. Note-se que a parte relacionada com a tradução manual está realçada por ser um processo complexo, como mencionado anteriormente.

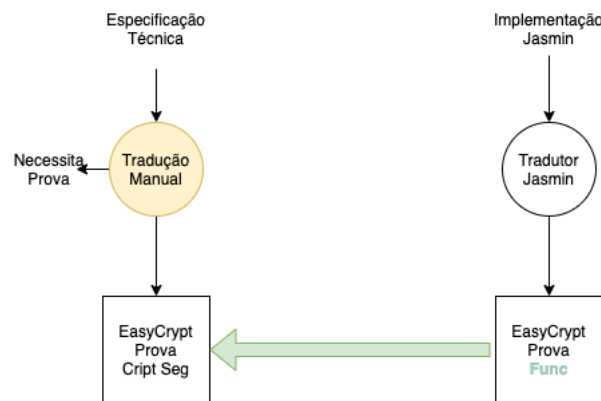


FIGURA 3.1: Processo de implementação/verificação de cifra em Jasmin e EasyCrypt.

Adicionalmente, a tradução do Jasmin para EasyCrypt gera uma tradução separada apenas para a verificação de *constant-time*. Esta prova é maioritariamente automática, sendo apenas necessário a prova de alguns lemas.

A semelhança da estrutura dos controlo de fluxos entre as duas linguagens faz com que a tradução seja directa [1]. Ainda assim, é preciso definir a semântica de todas as instruções x86 no EasyCrypt e, também, replicar o modelo de memória do Jasmin no EasyCrypt.

Foi usada a biblioteca do EC para construir novos tipos de dados. Em concreto, os tipos *Word* e *array* foram expandidos. Foram criados tipos *Word* com tamanhos definidos W_i com $i \in \{8, 16, 32, 64, 128\}$. Também foram expandidos os arrays, através do mesmo método. Foram criados *arrays* com tamanhos definidos e *arrays* de *words* também de tamanhos definidos. Assim, é possível fazer um mapeamento dos tipos dados usados pelo Jasmin no EasyCrypt. Por exemplo, tipos do Jasmin como o *u8, u16, u32...* são mapeados para *W8, W16, W32....*

```

1 op poly1305_pre (r : zp) (s : int) (m : Zp_msg)
2 (mem : global_mem_t) (inn, inl, kk : int) =
3   let body i =
4     let offset = i * 16 in
5     if i < size m -- 1
6     then load_block mem (inn+offset) 16
7     else load_block mem (inn+offset) (inl--ffset) in
8   let n = ceil (inl / 16) in
9   m = [body 0; ... ; body (n--)] ^
10  r = load_clamp mem kk ^
11  s = to_uint (loadW128 mem (kk + 16)).
12
13 op poly1305_loop (r : zp) (m : Zp_msg) (n : int) =
14   foldl (fun h i) (h + nth m i) * r) 0Zp [0;...;n--].
15
16 op poly1305_ref (r : zp) (s : int) (m : Zp_msg) =
17   let h' = poly1305_loop r m (size m) in
18   (((asint h') % 2128) + s) % 2128 .
19
20 op poly1305_post mem_pre mem_post outt rr ss mm =
21   mem_post =
22     storeW128 mem_pre outt (W128.of_int (poly1305_ref rr ss mm)).
23
24 op load_clamp(mem: global_mem_t) (ptr : address) =
25   let x = loadW128 mem ptr in
26   let xclamp =
27     x & (W128.of_int 0xFFFFFFFFC0FFFFFFC0FFFFFFC0FFFFFF) in
28   Zp.inzp (W128.to_uint xclamp).
29
30

```

FIGURA 3.2: Especificação do poly1305 em EasyCrypt [1]

Em suma, é possível entender como o EasyCrypt com o Jasmin fornecem uma plataforma para desenvolver implementações seguras e eficientes.

Capítulo 4

HACSpec

Neste capítulo introduz-se o projeto HACSpec e a sua relação com o EasyCrypt. Primeiramente, na secção 4.1, descreve-se o projeto e a linguagem HACSpec. Seguidamente, na secção 4.4 expõe-se a ligação do HACSpec com Easycrypt.

4.1 HACSpec

O HACSpec é um projeto colaborativo entre investigadores da Inria [19], Wire [20], Universidade do Porto, Aarhus University [21] e Max Planck Institute-Security and Privacy [22], com o objectivo de criar uma linguagem de alto nível capaz de desenvolver especificações de algoritmos criptográficas verificadas e formalmente provadas. Pretende-se criar uma linguagem com suficiente abstracção para que o criptógrafo comum possa desenvolver as suas próprias cifras sem se preocupar com instruções mais detalhadas para a máquina. O HACSpec contém quatro características que o destinge de outras linguagens de desenvolvimento de cifras:

1. É uma sub-linguagem de Rust, uma linguagem cada vez mais adoptada por programadores [23] [24] [25]. Assim, torna-se mais fácil a aprendizagem de HACSpec porque esta baseia-se na aprendizagem de Rust. Adicionalmente, como mencionado na secção 2.2, o Rust fornece um bom ponto de partida para o desenvolvimento de algoritmos criptográficos por obrigar o programador a programar utilizando um método seguro.
2. Contém uma biblioteca com funções e tipos de aritmética modular formalmente verificadas criando assim uma abstracção de alto nível para este tipo de operações.

Isto permite que o código em HACSpec fique semelhante ao pseudocódigo de algoritmos criptográficos padrão.

3. Os programas em HACSpec podem ser executados, criando , assim, a oportunidade de verificar e testar a implementação durante o desenvolvimento de uma implementação de uma cifra. O fato de serem executáveis, trás a vantagem de poderem servir como prototipo de uma cifra numa aplicação ainda em desenvolvimento.
4. Finalmente, a natureza funcional da sintaxe do HACSpec facilita a tradução para outras linguagens com propósito de verificação de corretude* e posteriormente de escrita de provas formais, linguagens como o F* [14] ou o Easycrypt.

Observa-se nas figuras 4.1 4.2 a semelhança entre o código da função poly em HACSpec e o pseudocódigo descritivo do algoritmo poly1305 no [rfc7539](#).

```

1 poly1305_mac(msg, key):
2     r = (le_bytes_to_num(key[0..15]))
3     clamp(r)
4     s = le_num(key[16..31])
5     accumulator = 0
6     p = (1<<130)-5
7     for i=1 upto ceil(msg length in bytes / 16)
8         n = le_bytes_to_num(msg[((i-1)*16)..(i*16)] | [0x01])
9         a += n
10        a = (r * a) % p
11    end
12    a += s
13    return num_to_16_le_bytes(a)
14 end
15

```

FIGURA 4.1: Pseudocódigo da função Poly, do poly1305[rfc7539](#).

*Optou-se traduzir o termo *correctness* para corretude.

```
1  pub fn poly(m: &ByteSeq, key: KeyPoly) -> Tag {
2      let r = le_bytes_to_num(&key.slice(0, BLOCKSIZE));
3      let r = clamp(r);
4
5      let s = le_bytes_to_num(&key.slice(BLOCKSIZE, BLOCKSIZE));
6      let s = FieldElement::from_secret_literal(s);
7
8      let mut a = FieldElement::from_literal(0u128);
9      for i in 0..m.num_chunks(BLOCKSIZE) {
10         let (len, block) = m.get_chunk(BLOCKSIZE, i);
11         let block_uint = le_bytes_to_num(&block);
12         let n = encode(block_uint, len);
13         a = a + n;
14         a = r * a;
15     }
16     poly_finish(a, s)
17 }
18
```

FIGURA 4.2: Implementação da função poly em HACSPec.

4.2 Linguagem

Um programa em HACSpec é um programa válido em Rust. A sintaxe do HACSpec é definida através de uma restrição de elementos da linguagem Rust* [26]. Identificam-se as restrições:

- O HACSpec não permite referências mutáveis(& mut t), só permite referências em argumentos de uma função. Esta definição deve-se ao fato de as declarações de estados mutáveis e imutáveis obrigarem o programador a pensar sobre invariantes e variantes. Isto não é típico de uma linguagem funcional.
- Não permite o uso do método clone() para fazer *deep copy* de valores de variáveis. Apenas o método copy() é permitido.
- Não permite qualquer tipo de vectores, listas e de slices a não ser Seqs e arrays e polinómios[†]. Restringe-se os tipos de dados disponíveis a um conjunto de tipos de dados definidos pelo HACSpec. Assim, é possível fixar um comportamento previsível sobre tipos de dados.

No apêndice A é possível observar a definição da linguagem.

4.3 Biblioteca

O HACSpec contém uma biblioteca com funções e tipos de dados úteis para desenvolver e implementar algoritmos criptográficos. De facto, é esta biblioteca que permite ao código em HACSpec ser semelhante ao pseudocódigo descritivo das cifras de *standards*. Além disso, também permite que o HACSpec se torne numa linguagem funcional. A biblioteca é constituída por *Secret Integers* e *Arrays* e *Corpos Finitos*. Será descrito em mais detalhe para melhor compreensão do que representam estes tipos.

- *Secret Integers*

Os *timing-attacks* [27] são ataques que analisam o tempo de execução das cifras, consoante o input. Se existir uma relação entre valores secretos (chaves números aleatórios) e o tempo de execução, um atacante poderá conseguir obter informação

*Sem contar com a biblioteca HACSpec, que traz novos tipos de dados e novas funções

[†]Esta estrutura de dados ainda está em desenvolvimento

através desta relação. Existem várias causas para esta vulnerabilidade. Por exemplo, existem algumas plataformas em que o tempo de execução de uma instrução de divisão de inteiros varia consoante o denominador e/ou o numerador [28]. As comparações entre inteiros também podem sofrer do mesmo problema mencionado anteriormente.

Para resolver este problema, o HACSpec introduz os *Secret Integers*, isto são inteiros em que o uso da divisão e da comparação não é permitida [26]. Estes inteiros (U8, U16, U32...) são equivalentes aos inteiros do Rust (u8, u16, u32...) mas com instruções específicas para o compilador quando são usados indevidamente. Cabe ao programador raciocinar quais valores são secretos através da especificação técnica. Estes valores secretos vão corresponder aos *Secret Integers*.

- *Arrays e Seq*

O HACSpec oferece dois tipos de vetores de tamanho fixo: *Arrays* e *Seqs*^{*}. Os *Seqs* podem ser vetores de qualquer tipo do HACSpec e permitem funcionalidades como indexação e *slicing*. Os *arrays*, definidos com o macro [31] *array!()*, são vetores de tamanho fixo definidos na altura da compilação. Por exemplo, a instrução *array!(Block, U8, 16)* cria um novo tipo com o nome "Block" que contém 16 posições do tipo U8. Os investigadores do HACSpec dão preferência ao uso de *arrays* em vez de *Seqs*. A principal razão para tal é a obrigação da declaração com o tamanho no caso dos *arrays* [26]. Assim, facilita a tradução para o EasyCrypt e o F*. No entanto, esta preferência não é clara, pois grande parte dos exemplos de cifras em HACSpec contêm *Seqs*. A designação de qual o tipo de *array* a usar ainda está em processo de discussão.

- *Corpos Finitos*

O HACSpec contém mecanismos para definir números inteiros em corpos finitos. Na figura 4.3 verifica-se a definição de um inteiro num corpo finito através da macro *public_nat_mod!*. O primeiro argumento define o nome do inteiro a ser usado. O segundo argumento não é usado mas, por causa de tecnicidades de implementação tem de ser definido o nome[†]. O terceiro argumento será o tamanho do corpo finito

^{*}Na verdade os *Seqs* não são propriamente vetores de tamanhos fixos. Os *Seqs* são *Vecs* do Rust que são vetores com a capacidade de bytes definida mas não tem de ter um tamanho fixo. [29] [30]

[†]O problema de implementação vem do uso de *macros*. O Rust não permite ter parâmetros de um objeto que não sejam argumentos da chamada de inicialização [31].

em bits. O quarto argumento será o valor do módulo do corpo finito*. Esta abstração de operações de matemática aritmética modular é uma das operações que torna HACSpec mais próximo ao pseudocódigo. Isto pode ser observado na figura 4.1.

```

1 public_nat_mod!(
2   type_name: FieldElement,
3   type_of_canvas: FieldCanvas,
4   bit_size_of_field: 131, // This amounts to 17 bytes
5   modulo_value: "03fffffffffffffffffffffffffffffb"
6 );
7

```

FIGURA 4.3: Definição de um inteiro num corpo finito em HACSpec.

4.4 HACSpec-Easycrypt

As especificações em HACSpec podem ser traduzidas para EasyCrypt automaticamente. Isto é realizado através de um tradutor que faz parte do HACSpec. A vantagem desta ligação entre as duas linguagens é que a tradução de HACSpec para EasyCrypt pode servir como uma prova de uma especificação técnica. Além disso, o EasyCrypt pode servir para verificar se o código HACSpec está correto.

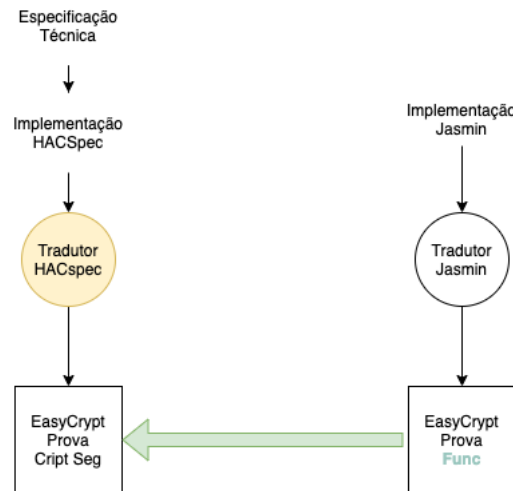


FIGURA 4.4: Processo de implementação/verificação de cifra em Jasmin e EasyCrypt com HACSpec.

*Este argumento é necessário porque os inteiros de corpos finitos em HACSpec são implementados com o *crate BigInt* e com um array. O *type_name* serve para definir o *BigInt* e o *type_of_canvas* serve para definir o array. *bit_size_of_field* serve para definir o tamanho do array.

Recorde-se a secção 3.3 em que é mencionado que "O processo de transposição é complexo. É necessário entrar em pormenores relacionados com os tipos de dados da especificação técnica. Este processo é demoroso e pode levar a erros". O HACSpec pode servir como uma implementação protótipo da especificação técnica, podendo até ser testado com vetores de teste fornecidos pela especificação, e depois ser traduzido automaticamente para EasyCrypt. Note-se na figura 4.4 a demonstração do processo mencionado na secção 3.3, mas agora com o HACSpec incluído.

Capítulo 5

Desenvolvimento

Neste capítulo descreve-se a concepção e implementação do tradutor. Na secção 5.1 expõe-se a concepção e apresenta-se o estado do tradutor. De seguida, na secção 5.2, é retratado o processo de implementação.

5.1 Concepção

Antes de se ter iniciado o desenvolvimento do tradutor no contexto da presente dissertação, já existia uma versão do mesmo. No entanto, este tradutor não produzia um código com a semântica correta. Adicionalmente, a utilização da biblioteca e do tipos de dados do EasyCrypt não era a correta. Definiu-se como objectivo alterar o tradutor para produzir um código com a semântica correta. Essencialmente, isto significa que o código produzido tem de ser o mais semelhante possível com o original, preservando todos os tipos e nomes de função. Recorde-se o que foi referido na secção 3.3: "Além disso esta tradução tem de ser o mais exacto possível com a especificação". Apresenta-se, de seguida, a arquitectura do tradutor com o objetivo de contextualizar e depois descreve-se em detalhe as alterações necessárias ao tradutor para cumprir o objetivo proposto.

5.1.1 Arquitetura do Tradutor

A linguagem HACSpec é uma sub-linguagem do Rust logo, o compilador está implementado em Rust e toma proveito da API disponibilizada, `rustc_driver` [26]. A figura 5.2 demonstra os passos de compilação. Primeiro, o código em HACSpec é compilado pelo compilador de Rust que gera a árvore sintáctica abstracta de Rust [26]. De seguida, esta

árvore é traduzida para uma AST representativa de um conjunto de linguagem mais restrito, o HACSpec, gerando, assim, a árvore AST de HACSpec. Neste passo são apanhados *tokens* que não pertencem à linguagem de HACSpec. Na figura 5.1 pode-se observar um exemplo de mensagem de erro do compilador. A mensagem de erro é relativa ao uso dos vectores nativos do Rust(&[U8]), os quais o HACSpec não permite.

```

1  error[Hacspec]: type not allowed in Hacspec
2  --> /Users/raulmendiratta/Documents/Universidade/TESE/projects/
   hacspec_test/hacspec/hacspec/examples/chacha20/src/chacha20.rs:117:37
3  |
4  117 | pub fn chacha(key: Key, iv: IV, m: &[U8]) -> ByteSeq {
5  |                                     ^^^^
6
7  error: unable to translate to Hacspec due to out-of-language errors
8

```

FIGURA 5.1: Mensagem de erro do compilador de HACSpec.

Ainda assim, existem alguns *tokens* que passam. O Typechecker vai apanhar todos os *tokens* que conseguiram passar na fase anterior. Estes *tokens* que passam são *tokens* que remetem a invocações a funções de bibliotecas externas ao HACSpec e declarações com inferência de tipos [26]. Finalmente, a AST em HACSpec é traduzida para um código em EasyCrypt, sendo neste passo que a maior parte do trabalho relativo à presente investigação foi realizada.

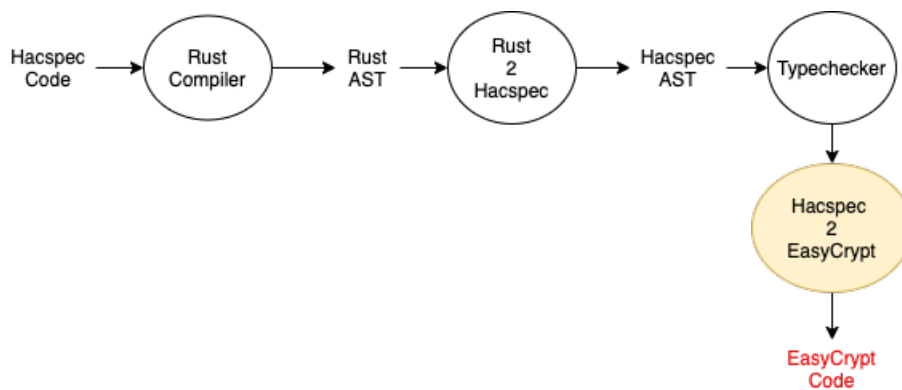


FIGURA 5.2: Arquitetura do Tradutor.

5.1.2 Alterações necessárias

Imports O tradutor inicial produzia uma tradução em que no início do código continha instruções *import* para todas as bibliotecas e tipos do easycrypt, ainda que não fossem usados. Na figura 5.3 é possível observar parte do código produzido pelo tradutor do poly1305. Repare-se que na linha 2 do código existe uma importação de todos os tipos de *array* e isto introduzia informação desnecessária no código. Pretende-se que o tradutor insira na tradução apenas imports que sejam necessários.

```

1 require import List Int IntDiv CoreMap AllCore.
2 require import Array3 Array4 Array8 Array12 Array16 Array17 Array32 Array64
3   .
4   require import WArray64.
5   from Jasmin require import JModel JMemory JArray.
6   require import HacsSpec.
7   ...
8
```

FIGURA 5.3: Parte de código do Poly1305 gerado pelo tradutor.

Declarações Na tradução original, as declarações de variáveis globais eram feitas no início do código. Decidiu-se passar estas declarações para o ficheiro de Prelúdio. Criando, assim, mais legibilidade no ficheiro principal.

snake_case O tradutor estava a traduzir o código para a convenção de nomes *snake_case* [32]. Repare-se na figura 5.4 o tipo *ByteSeq* do HACSpec traduzido para *byte_seq*. Optou-se de deixar a convenção de nomes original do HACSpec. Como o HACSpec é uma sub-linguagem do Rust [26] a convenção de nomes vai ser a mesma que a do Rust que se pode observar na tabela 5.1 retirada do manual do Rust [33].

```

1 op le_bytes_to_num (b_0 : byte_seq) ...
2 ...
3
```

FIGURA 5.4: Parte de código do Poly1305 gerado pelo tradutor.

Item	Convenção
Modules	snake_case
Types	UpperCamelCase
Traits	UpperCamelCase
Enum variants	UpperCamelCase
Functions	snake_case
Methods	snake_case
Macros	snake_case!
Local variables	snake_case
Statics	SCREAMING_SNAKE_CASE
Constants	SCREAMING_SNAKE_CASE

TABELA 5.1: Tabela de convenção de nomes do HACSpec. [33]

Tipos e Nomes de Função Os tipos e os nomes de função do HACSpec estavam a sofrer uma alteração durante a tradução. Na figura 5.5, repara-se na linha 1 do código em HACSpec, o tipo *U128* é traduzido para *uint128*.

Note-se na linha 2, a tradução da função *U128::Wordfrom_slice* para *array_16_from_slice*.

Pretende-se alterar esta tradução com o objectivo de preservar a nomenclatura das funções e dos tipos.

```

1 pub fn le_bytes_to_num(b: &
  ByteSeq) -> U128 {
2   let block_as_u128 = U128Word
    ::from_slice(b, 0, min(
      BLOCKSIZE, b.len()));
3   U128_from_le_bytes(
    block_as_u128)
4 }
5

```

```

1 op le_bytes_to_num (b_0 :
  byte_seq) : uint128 =
2   let block_as_u128_1 =
3     array_16_from_slice (secret (
      pub_u8 8)) (b_0) (0) (
4       min (blocksize) (seq_len (
        b_0)))
5   in
6     uint128_from_le_bytes (
      block_as_u128_1).
7

```

FIGURA 5.5: Função *le_bytes_to_num* em HACSpec, à esquerda, e em EasyCrypt à direita

Corpos Finitos A tradução da declaração de corpos finitos era errada. O valor semântico perde-se, sendo que o tradutor para os corpos finitos não produz um código semanticamente equivalente. Repara-se na figura 5.6 que na tradução o *field_element* está declarado como *int* sem ter qualquer tipo de relação com o *field_canvas*. Recorde-se na secção 4.3 em que foi afirmado que: "O terceiro argumento será o tamanho do corpo finito em bits. O quarto argumento será o valor do modulo do corpo finito". Esta informação não

está a ser preservada na tradução. Pretende-se, portanto, alterar o tradutor para traduzir corretamente os corpos finitos.

```

1 public_nat_mod!(
2   type_name: FieldElement,
3   type_of_canvas: FieldCanvas,
4   bit_size_of_field: 131, //
   This amounts to 17 bytes
5   modulo_value: "03
   ffffffffffffffffffffffffffffffffffb
   "
6 );
7

```

```

1 type field_canvas = pub_uint8
   Array17.t.
2
3 type field_element = int.
4

```

FIGURA 5.6: Declaração de corpos finitos, à esquerda em HACSpec, e em EasyCrypt à direita.

Funcionamento do Tradutor A figura 5.7 mostra o funcionamento do tradutor através do exemplo da tradução da função *U128Word::from_seq*. Esta função é traduzida para a função *array_16_from_seq* que vai conter a sua definição num ficheiro em separado. Este ficheiro tem o propósito de funcionar como biblioteca, conforme indicado na figura 5.7. O compilador do EasyCrypt lê a função do código produzido e procura a sua definição no ficheiro que a contém (neste caso os investigadores chamam-lhe Hacspect.ec). Esta abordagem torna mais fácil acrescentar definições das funções traduzidas. Como são traduzidas todas as funções *T::from_seq* para *array_16_from_seq*, desde que *T* seja traduzível para *array_16*, só será necessário escrever uma definição, que é a definição de *array_16_from_seq*.

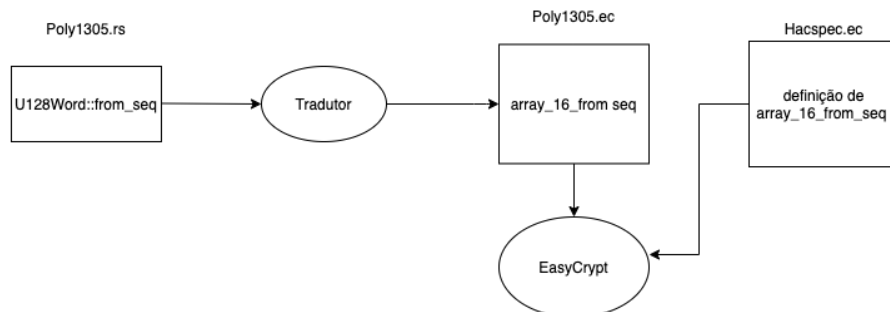


FIGURA 5.7: Workflow do tradutor

Pretende-se conceber um novo método para o funcionamento do tradutor com o objetivo de conservar os nomes. A consequência de conservar os nomes é que vai ser preciso a

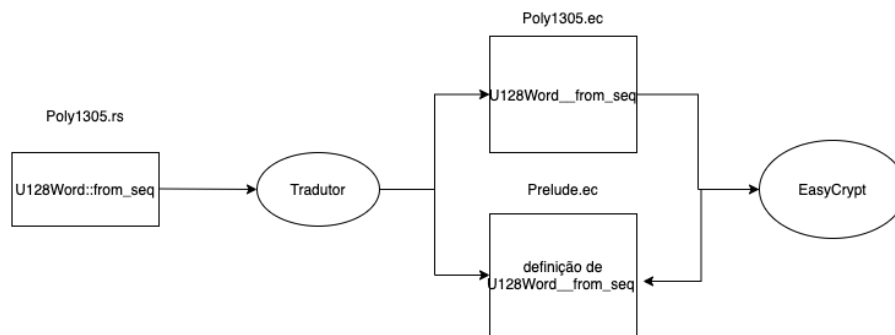
```

1   op array_16_from_seq (x: 'a Sequence.t) : 'a Array16.t.
2

```

FIGURA 5.8: Definição de `array_16_from_seq` in EasyCrypt

sua definição. Para isso, o tradutor, para além de gerar a tradução, tem de gerar também um ficheiro prelúdio que vai conter as definições de tipos e de nomes de funções, para as quais se decidiu conservar o nome, necessárias para o EasyCrypt interpretar o código. Na figura 5.9 é possível visualizar o diagrama do funcionamento pretendido do tradutor para a função `U18Word::from_seq`. Note-se que o nome da função se mantém.

FIGURA 5.9: *Workflow* do tradutor.

5.2 Implementação

Descreve-se, agora, em detalhe as alterações realizadas para cumprir o objetivo proposto mencionado anteriormente.

Imports Para criar uma tradução em que só importa os *arrays* necessários foi preciso criar uma função que tem como input a AST de HACSpec e que verifica quais *arrays* são necessários importar. Repare-se na figura 5.10, na linha 6 e 7 a instrução `write!(....`. Esta instrução está a escrever a *string Arrays* concatenada com uma *string* que vai ter o tamanho do *array*. Este tamanho é capturado nas declarações de `arrays(Item::ArrayDecl(...)`.

Declarações e Funcionamento do Tradutor Com o objetivo de alterar o tradutor para que transfira as declarações para um ficheiro de prelúdio, dividiu-se os items da AST entre duas funções. Uma função recebe os items com as funções, a outra recebe as declarações de *arrays*, constantes e corpos finitos. Cada função imprime num ficheiro diferente

criando, assim, um ficheiro de prelúdio e um ficheiro principal. Repare-se na figura 5.11, na linha 5, a função *write_prelude_dcl* que captura o item *Item::ArrayDecl*.

```

1      ...
2      Item::ArrayDecl(_name, size, _cell_t, _index_typ) => {
3          match size.0 {
4              Expression::Lit(Literal::Usize(x)) => {
5                  let size = format!("{}",x);
6                  if !flag_list.contains(&size){
7                      write!(file_pre, "_Array{}",size).unwrap();
8                      write!(file_main, "_Array{}",size).unwrap();
9                      flag_list.push_str(&size);
10                 ...
11

```

FIGURA 5.10: Parte do código do tradutor.

```

1 fn translate_item<'a>(i: &'a Item, typ_dict: &'a TypeDict) -> Option<RcDoc
2     <'a, ()>> {
3     match i {
4         ...
5         Item::ArrayDecl(_,_,_,_) => {
6             write_prelude_dcl(Some(i),&typ_dict);
7             None
8             //This goes to the prelude file now
9         },
10     ...
11     fn write_prelude_dcl<'a>(i: Option<&'a Item>, typ_dict: &'a TypeDict){
12     ...

```

FIGURA 5.11: Parte do código do tradutor.

snake_case A alteração do tradutor relativamente ao *snake_case* foi elementar. O código do tradutor tinha uma função que estava a produzir código em *snake_case*. A alteração foi apenas retirar o uso dessa função.

Tipos e nomes de função Como foi mencionado anteriormente, pretende-se criar um prelúdio com as definições das funções (traduzidas para operadores). Criaram-se traduções modelo de definições de funções. Por exemplo pode-se definir uma tradução modelo para as funções *T.slice*. Para tal, em vez de criar uma definição para cada diferente tipo *T*, cria-se um modelo de tradução e apenas substitui-se as partes necessárias. Observe-se a

figura 5.12 na qual é criado um modelo para a definição do operador T_{slice} . Note-se na linha 9,10,11 em que se concatena *op* com uma variável *string* que vai conter o nome de *T*. Ao todo foi necessário criar 17 modelos diferentes.

```

1 fn write_prelude_fcall<'a>(module_name: &str, func_ident: &str, typ_str:
  Option<&str>){
2   ...
3   array if array == array_module.as_str() => {
4     match (func_ident,seq) {
5       ...
6       ("slice",_) => {
7         ...
8         RcDoc::as_string("op_")
9         .append(RcDoc::as_string(array_module.to_string()))
10        .append(RcDoc::as_string("_slice_(s:_a_Array)")
11        .append(RcDoc::as_string(array_info.0.clone()))
12        ...
13

```

FIGURA 5.12: Parte do código do tradutor.

Corpos Finitos Alterou-se o tradutor para produzir a tradução correta da declaração dos corpos finitos do HACSpec. Na figura 5.13 é possível observar a tradução correta. Note-se que esta tradução usa os corpos finitos da biblioteca do Jasmin para EasyCrypt, *ZModP*. Com o uso de *ZmodP* é possível preservar o valor semântico.

```

1 require import ZModP.
2
3 clone import ZModField as ZModField_FieldElement with
4   op p <- 1361129467683753853853498429727072845819
5   rename "zmod" as "FieldElement".
6

```

FIGURA 5.13: Tradução correta da declaração de corpos finitos.

Capítulo 6

Resultados e Validação

Neste capítulo introduz-se os resultados e validação. Primeiro apresenta-se a metodologia usada para a validação. Na secção 6.1 descreve-se os resultados e a validação.

Metodologia de Validação Para validar a tradução definida optou-se por realizar uma prova preliminar de equivalência entre a tradução pretendida e uma prova de existente do poly1305. Valida-se o tradutor verificando se está a gerar as traduções de acordo com a tradução definida. Para além do poly1305 também foi usado o chacha20 como *input* teste para verificar a tradução.

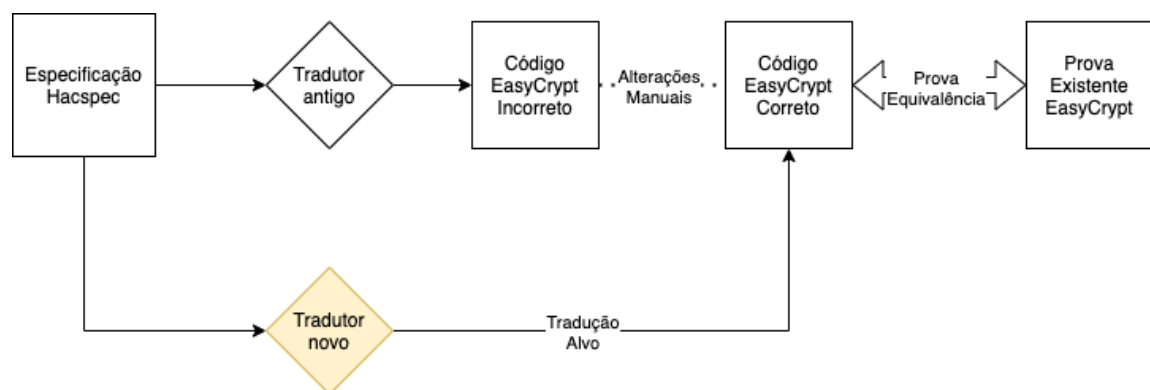


FIGURA 6.1: Processo Validação

6.1 Resultados

Nesta secção são indicadas as novas traduções a serem verificadas pelo EasyCrypt. Em concreto as traduções dos tipos e nomes de função. Note-se que só são apresentados

alguns exemplos de *inputs* e de *outputs*. Descrever todos os *inputs* e *outputs* possíveis seria muito extensivo. As alterações ao tradutor relacionadas com a relocalização dos *Imports* e das declarações são automaticamente validadas pois apenas alteram o aspecto e a organização do código. São apresentados *screenshots* do programa do EasyCrypt a validar partes das traduções. Antes de descrever os resultados é necessário apresentar duas notas para compreender as figuras:

1. Quando o EasyCrypt valida uma linha de código, o fundo do editor de texto nessa linha, neste caso o Emacs, aparece a azul.
2. As figuras contêm um código em HACSpec à esquerda e o screenshot da tradução em EasyCrypt à direita.

As traduções completas podem ser consultadas no Anexo B.

Declaração e Funcionamento do Tradutor Na figura 6.2 e na figura 6.3 é possível observar a tradução de declaração de tipos a ser validada pelo EasyCrypt. Repare-se que o nome dos tipos não é alterado. Note também que as traduções das declarações usam os tipos nativos do HACSpec (*U8*, *usize*,...). Note-se também que estas declarações estão escritas num ficheiro separado chamado *Prelude.ec*.

```

1 bytes!(KeyPoly, 32);
2
3 const BLOCKSIZE: usize =
4   16;
5 bytes!(Block, 16);
6
7 public_bytes!(Tag, 16);
8 ...
9

```

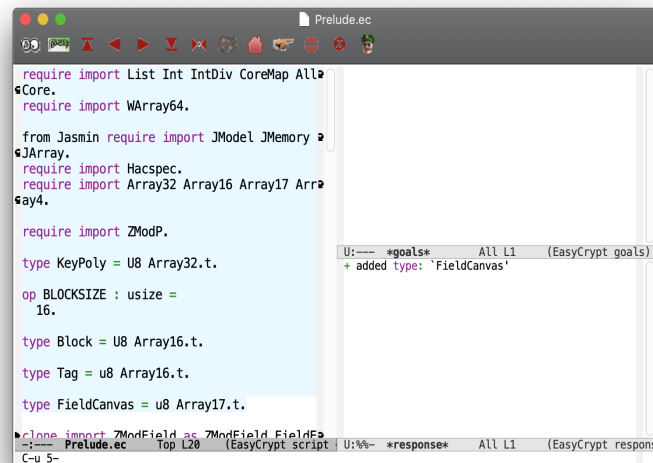


FIGURA 6.2: Tradução de declaração de tipos.

Observe-se, na figura 6.3 a tradução da instrução da linha 4 do código em HACSpec para EasyCrypt. Na tradução esta declaração é correctamente separada em duas, na declaração do *State* e na declaração do *StateIdx*.

```

1 // Import hacspect and all
  needed definitions.
2 use hacspect_lib::*;
3
4 array!(State, 16, U32,
  type_for_indexes:
  StateIdx);
5 bytes!(StateBytes, 64);
6 bytes!(IV, 12);
7 bytes!(Key, 32);
8

```

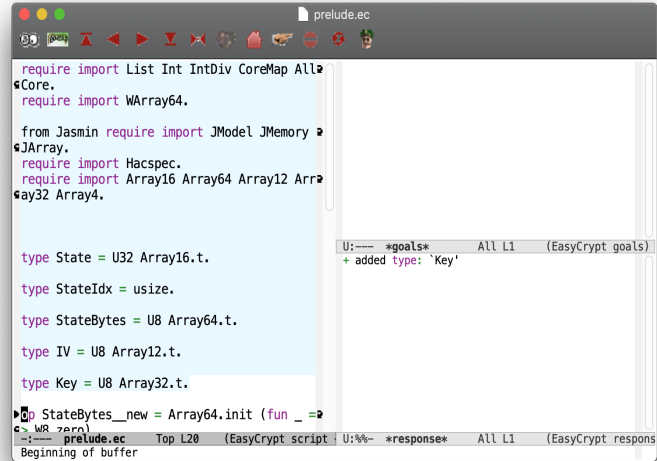


FIGURA 6.3: Tradução de declaração de tipos.

Tipos e nomes de Funções Observe-se na figura 6.4 a tradução da funções *a.to_public_byte_seq_be().slice*. Repare que foi apurado o tipo de *a* e traduzido para *ByteSeq_slice (FieldElement_to_public_byte_seq_be....* Os nomes das funções foram preservados. Na figura 6.5 é possível ver as definições dos operadores *ByteSeq_slice* e *FieldElement_to_public_byte_seq_be*. Recorde-se que estas definições são geradas pelo tradutor. Na figura 6.5 pode-se ver o uso dos *templates* como mencionado na secção 5.2. Repare-se nas definições dos operadores *u128Word__from_seq* e *Tag__from_seq*. A definição do operador é igual, excepto o nome. Isto também é possível observar nas definições dos operadores *ByteSeq_slice* e *KeyPoly_slice templates*.

```

1 ...
2 let a = a.
  to_public_byte_seq_be
  ().slice(1, BLOCKSIZE
  );
3 let a =
  u128_from_be_bytes(
  u128Word::from_seq(&a
  ));
4 ...
5 Tag::from_seq(&
  u128_to_le_bytes(a))
6 ...
7

```

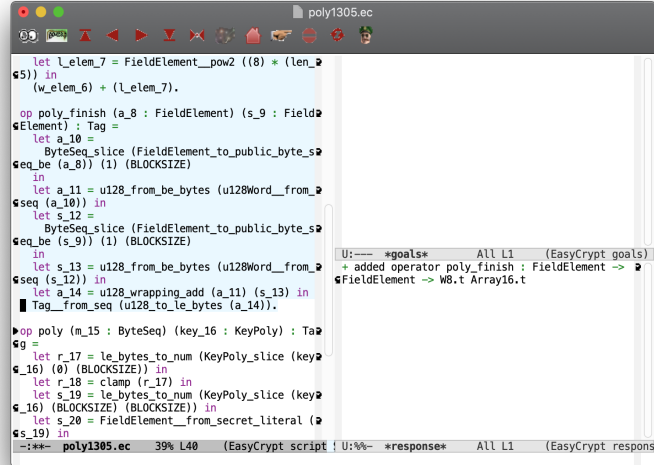


FIGURA 6.4: Tradução de invocações de funções.

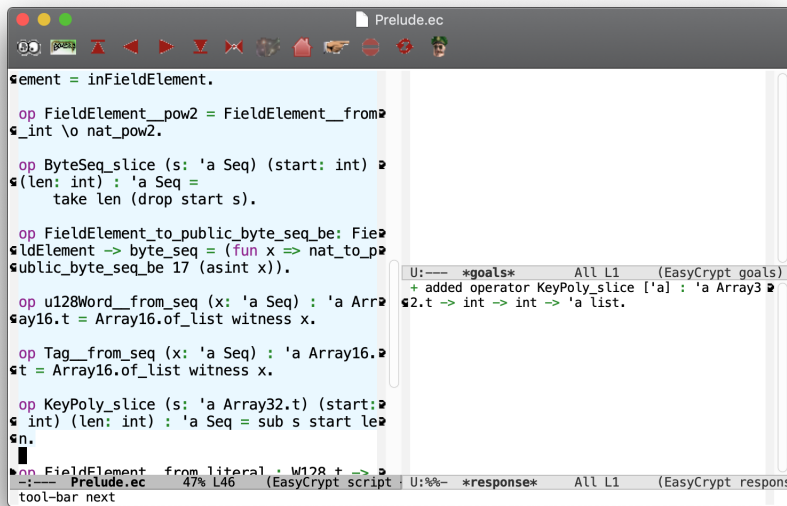


FIGURA 6.5: Declaração de operadores.

Corpos Finitos Note-se na figura 6.6 a tradução da declaração de um corpo finito (*FieldElement*). Este é traduzido mantendo o nome original *FieldElement*. Repare-se que o tradutor está a gerar a tradução correta como mencionado na secção 5.2.

```

1 public_nat_mod!(
2   type_name:
3   FieldElement,
4   type_of_canvas:
5   FieldCanvas,
6   bit_size_of_field:
7   131, // This amounts
      to 17 bytes
      modulo_value: "03
      ffffffffffffffffff..."
8 );

```

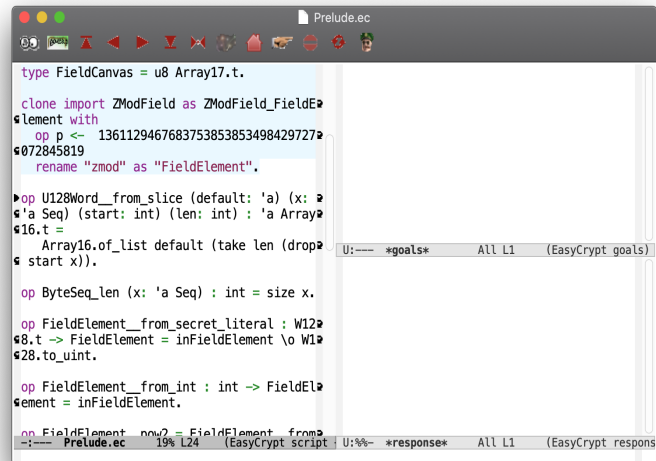


FIGURA 6.6: Tradução de declaração de tipos.

Discussão Os exemplos de tradução demonstram a capacidade do tradutor produzir uma tradução válida. Em concreto, a tradução do poly1305 é totalmente verificada correta no EasyCrypt. A tradução do chacha20 é parcialmente verificada correta, sendo que o tradutor necessita ainda de desenvolvimento. Isto será discutido com mais detalhe no próximo capítulo.

Capítulo 7

Conclusão

Neste capítulo, apresenta-se a conclusão. Começa-se, por apresentar uma breve prospectiva sobre o âmbito da presente dissertação. De seguida, descreve-se o trabalho realizado. Finalmente, indica-se o trabalho a desenvolver no futuro.

Prospectiva Os projetos HACSpec e Jasmin são dois projetos promissores para o desenvolvimento e implementação, respectivamente. Espera-se que o projeto HACSpec se torne numa plataforma capaz de ligar várias linguagens com diferentes níveis de abstracção podendo, assim, ajudar a aproximar a especificação técnica com a implementação. Adicionalmente, poderá criar um ponto central que junta criptógrafos, investigadores de implementações de cifras, e verificadores formais de criptografia. Em prospectiva, o HACSpec pode ser referenciado nas especificações padrão. Por exemplo, imagine-se que existe uma nova especificação de uma nova cifra para qual ainda não existe nenhuma implementação otimizada, essa especificação pode ter um link para uma implementação em HACSpec. Assim, um programador poderá consultar a implementação em HACSpec e usá-la como referência para uma implementação em Jasmin, depois poderá usar o tradutor para EasyCrypt para verificar e provar que a implementação está correta.

Trabalho Realizado Nesta dissertação foi possível alterar um tradutor de HACSpec para EasyCrypt. Desenvolveu-se a capacidade de validar este tradutor verificando que se consegue ligar uma prova existente com uma especificação HACSpec. Especificamente, alterou-se com sucesso um tradutor que produz uma tradução correta do poly1305 e, em parte, do chacha20. Não foi possível acabar o desenvolvimento do tradutor para o chacha20. De facto, inicialmente, em âmbito da disciplina de projeto de dissertação,

foi proposto como objetivo o tradutor funcionar para mais exemplos como o SHA-3 e o curve25519. No entanto, no início do desenvolvimento do tradutor entendeu-se que este objetivo não iria ser cumprido. No início do desenvolvimento, dedicou-se muito tempo a entender o HACSpec. Note-se que, como é uma tecnologia em desenvolvimento, não há disponível muita informação como documentação, *papers*, *web-sites*, etc. De facto, o único conteúdo disponível é o repositório *online* e o contacto direto com os investigadores do projeto. Além disso, admite-se que houve um processo de aprendizagem da linguagem Rust, que é notória por ser uma linguagem com uma grande curva de aprendizagem. Adicionalmente, também foi necessário compreender o EasyCrypt e o Jasmin. Em suma, a aprendizagem e adaptação ao âmbito tecnológico da presente dissertação, teve um impacto no tempo necessário para o desenvolvimento. Ainda assim, foi possível criar um tradutor que produz uma tradução válida para o poly1305.

Trabalho Futuro Como mencionado anteriormente, é necessário continuar o desenvolvimento do tradutor. Idealmente, o tradutor produz traduções válidas para todos os exemplos de implementações em HACSpec. Em concreto, é preciso criar traduções válidas para todas as funções e tipos da biblioteca do HACSpec.

Bibliografia

- [1] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub, “The last mile: High-assurance and high-speed cryptographic implementations,” *CoRR*, vol. abs/1904.04606, 2019. [Online]. Available: <http://arxiv.org/abs/1904.04606> [Cited on pages xv, 8, 14, 15, and 23.]
- [2] D. J. Bernstein, “The poly1305-aes message-authentication code,” in *Fast Software Encryption*, H. Gilbert and H. Handschuh, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 32–49. [Cited on page 8.]
- [3] *Rust Wiki*. [Online]. Available: [https://en.wikipedia.org/wiki/Rust_\(programming_language\)](https://en.wikipedia.org/wiki/Rust_(programming_language)) [Cited on page 10.]
- [4] *The Rust Programming Language*. [Online]. Available: <https://doc.rust-lang.org/book/> [Cited on page 12.]
- [5] Writing high-speed software. [Online]. Available: <https://cr.yp.to/qhasm.html> [Cited on page 13.]
- [6] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, “Jasmin: High-Assurance and High-Speed Cryptography,” in *CCS 2017 - Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, United States, Oct. 2017, pp. 1–17. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-01649140> [Cited on pages 13, 14, and 15.]
- [7] “Jasmin wiki.” [Online]. Available: <https://github.com/jasmin-lang/jasmin/wiki> [Cited on page 14.]
- [8] C. Baritel-Ruet, “Formal security proofs of cryptographic standards,” Ph.D. dissertation, 10 2020. [Cited on pages 17 and 18.]

- [9] H. Geuvers, “Proof assistants: History, ideas and future,” *Sadhana*, vol. 34, no. 1, pp. 3–25, Feb 2009. [Online]. Available: <https://doi.org/10.1007/s12046-009-0001-5> [Cited on page 17.]
- [10] The coq proof assistant. [Online]. Available: <https://coq.inria.fr/> [Cited on page 17.]
- [11] Agda. [Online]. Available: <https://wiki.portal.chalmers.se/agda/pmwiki.php> [Cited on page 17.]
- [12] Lean. [Online]. Available: <https://leanprover.github.io/> [Cited on page 17.]
- [13] Prototype verification system. [Online]. Available: <https://pvs.csl.sri.com/> [Cited on page 17.]
- [14] N. Swamy, C. Hrițcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, “Dependent types and multi-monadic effects in f^* ,” *SIGPLAN Not.*, vol. 51, no. 1, p. 256–270, Jan. 2016. [Online]. Available: <https://doi.org/10.1145/2914770.2837655> [Cited on pages 17 and 26.]
- [15] Easycrypt: Computer-aided cryptographic proofs. [Online]. Available: <https://www.easycrypt.info/trac/> [Cited on pages 17 and 18.]
- [16] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, “Computer-aided security proofs for the working cryptographer,” in *Advances in Cryptology – CRYPTO 2011*, P. Rogaway, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 71–90. [Cited on page 18.]
- [17] *EasyCrypt Reference Manual*, 2018. [Online]. Available: <https://www.easycrypt.info/documentation/refman.pdf> [Cited on pages 18 and 19.]
- [18] M. Barbosa. Easycrypt and jasmin tutorial. [Online]. Available: <https://www.easycrypt.info/JasminECTutorial.mp4> [Cited on page 21.]
- [19] Inria. [Online]. Available: <https://www.inria.fr/fr> [Cited on page 25.]
- [20] Wire. [Online]. Available: <https://wire.com/en/> [Cited on page 25.]
- [21] Aarhus university. [Online]. Available: <https://www.au.dk/> [Cited on page 25.]

- [22] Max plack institute- security and privacy. [Online]. Available: <https://www.cis.mpg.de/> [Cited on page 25.]
- [23] J. M. Perkel, "Why scientists are turning to rust," *Nature*, vol. 588, pp. 185–186, 2020. [Online]. Available: <https://doi.org/10.1038/d41586-020-03382-2> [Cited on page 25.]
- [24] M. Asay, "Why the rust language is on the rise," *InfoWorld*, 2019. [Online]. Available: <https://www.infoworld.com/article/3451296/why-the-rust-language-is-on-the-rise.html> [Cited on page 25.]
- [25] B. Vigliarolo, "The rust programming language now has its own independent foundation," *TechRepublic*, 2021. [Online]. Available: <https://www.techrepublic.com/article/the-rust-programming-language-now-has-its-own-independent-foundation/> [Cited on page 25.]
- [26] D. Merigoux, F. Kiefer, and K. Bhargavan, "Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust," Inria, Technical Report, Mar. 2021. [Online]. Available: <https://hal.inria.fr/hal-03176482> [Cited on pages 28, 29, 33, 34, and 35.]
- [27] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005, web Security. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1389128605000125> [Cited on page 28.]
- [28] *Why Constant-Time Crypto?* [Online]. Available: <https://www.bearssl.org/constanttime.html> [Cited on page 29.]
- [29] *Seq.rs*. [Online]. Available: <https://github.com/hacspec/hacspec/blob/2395da75c465080d12944fb35de31da671a8f372/lib/src/seq.rs> [Cited on page 29.]
- [30] *Vec*. [Online]. Available: <https://doc.rust-lang.org/src/alloc/vec/mod.rs.html> [Cited on page 29.]
- [31] *Macros*. [Online]. Available: <https://doc.rust-lang.org/book/ch19-06-macros.html> [Cited on page 29.]
- [32] Snake case. [Online]. Available: https://en.wikipedia.org/wiki/Snake_case [Cited on page 35.]

- [33] *Rust API Guidelines: Naming*. [Online]. Available: <https://rust-lang.github.io/api-guidelines/naming.html> [Cited on pages 35 and 36.]
- [34] hacspec language. [Online]. Available: <https://github.com/hacspec/hacspec/blob/master/Language.md> [Cited on page 54.]

Apêndice A

Definição da linguagem HACSpec

```
1 Items i :=
2     | use u; (* import modules *)
3     | const x : tau = c; (* constant variables *)
4     | type y = tau'; (* type aliases *)
5     | array!(y, tau, c); (* fixed-length array declaration *)
6     | poly!(y', y, a); (* fixed-length polynomials *)
7     | field_integers!(y, c, c); (* abstract field integer declaration *)
8     | fn f([x: (&)tau,]+) -> tau' { e } (* function *)
9     | enum y { [z(tau),]+ } (* enum type *)
10    | struct y { [f: tau,]+ } (* struct type *)
11
12 Use path u :=
13     | [m::]*m' (* sequence of nested modules*)
14
15 Type tau :=
16     | bool | usize
17     | u8 | u16 | u32 | u64 | u128 | U8 | U16 | U32 | U64 | U128
18     | i8 | i16 | i32 | i64 | i128 | I8 | I16 | I32 | I64 | I128
19     | Seq<tau> (* Unknown-length array *)
20     | y (* type variable *)
21     | ([tau,]+) (* tuples *)
22
23 Statement s :=
```

```

24 | let (mut) p (: tau) = e (* let binding *)
25 | x = e (* mutable variable reassignment *)
26 | if e1 { e2 } (else { e3 }) (* if statement | Maybe
27 we want this to be an expression? *)
28 | for x in e1..e2 { s } (* for loop *)
29 | s1; s2 (* sequencing *)
30 | x[e1] = e2 (* array update *)
31
32 Expression e :=
33 | l (* literal *) | (u::)x (* variable *)
34 | (u::)(y::)f([(&)e]+); (* function call *)
35 | e.f([e']+) (* method call *)
36 | ([e]) (* tuple *)
37 | e1.n (* tuple member access *)
38 | e1..e2 (* range *)
39 | e1 op e2 (* arithmetic operations *) | unop e (* unary op *)
40 | x[e] (* array indexing *)
41 | e1 as e2 (* UNSafe integer casting *)
42
43 Operator op := + | - | * | / | ^ | && | || | & | | | % | >> | << |
44 | == | != | <= | >=
45 Unary operator unop := ~ | ! | -
46
47 Pattern p :=
48 | x
49 | ([p,]*) (* tuple destructuring *)
50 | _ (* wildcard *)
51

```

CÓDIGO A.1: Definição da linguagem Hacspecc.[\[34\]](#)

Apêndice B

Traduções geradas.

```
1 require import Prelude Hacspec.
2 require import List Int IntDiv CoreMap AllCore.
3 require import WArray64.
4
5 from Jasmin require import JModel JMemory JArray.
6 require import Hacspec.
7 require import Array32 Array16 Array17 Array4.
8
9 require import ZModP.
10 import ZModField_FieldElement.
11
12 op le_bytes_to_num (b_0 : ByteSeq) : U128 =
13   let block_as_u128_1 =
14     U128Word__from_slice (secret (u8 8)) (b_0) (0) (
15       min (BLOCKSIZE) (ByteSeq_len (b_0)))
16   in
17   U128_from_le_bytes (block_as_u128_1).
18
19 op clamp (r_2 : U128) : FieldElement =
20   let r_uint_3 =
21     W128.andw (r_2) (secret (u128 21267647620597763993911028882763415551))
22   in
23   FieldElement__from_secret_literal (r_uint_3).
```

```

24
25 op encode (block_uint_4 : U128) (len_5 : usize) : FieldElement =
26   let w_elem_6 = FieldElement__from_secret_literal (block_uint_4) in
27   let l_elem_7 = FieldElement__pow2 ((8) * (len_5)) in
28   (w_elem_6) + (l_elem_7).
29
30 op poly_finish (a_8 : FieldElement) (s_9 : FieldElement) : Tag =
31   let a_10 =
32     ByteSeq_slice (FieldElement_to_public_byte_seq_be (a_8)) (1) (BLOCKSIZE
33       )
34   in
35   let a_11 = u128_from_be_bytes (u128Word__from_seq (a_10)) in
36   let s_12 =
37     ByteSeq_slice (FieldElement_to_public_byte_seq_be (s_9)) (1) (BLOCKSIZE
38       )
39   in
40   let s_13 = u128_from_be_bytes (u128Word__from_seq (s_12)) in
41   let a_14 = u128_wrapping_add (a_11) (s_13) in
42   Tag__from_seq (u128_to_le_bytes (a_14)).
43
44 op poly (m_15 : ByteSeq) (key_16 : KeyPoly) : Tag =
45   let r_17 = le_bytes_to_num (KeyPoly_slice (key_16) (0) (BLOCKSIZE)) in
46   let r_18 = clamp (r_17) in
47   let s_19 = le_bytes_to_num (KeyPoly_slice (key_16) (BLOCKSIZE) (BLOCKSIZE
48     )) in
49   let s_20 = FieldElement__from_secret_literal (s_19) in
50   let a_21 = FieldElement__from_literal (u128 0) in
51   let a_21 =
52     foldi (0) (ByteSeq_num_chunks (m_15) (BLOCKSIZE)) (fun i_22 a_21 =>
53       let (len_23, block_24) = ByteSeq_get_chunk (m_15) (BLOCKSIZE) (i_22)
54       in
55       let block_uint_25 = le_bytes_to_num (block_24) in
56       let n_26 = encode (block_uint_25) (len_23) in

```

```
53     let a_21 = (a_21) + (n_26) in
54     let a_21 = (r_18) * (a_21) in
55     a_21)
56   a_21
57 in
58 poly_finish (a_21) (s_20).
```

CÓDIGO B.1: Código da tradução do poly1305.

```
1   require import List Int IntDiv CoreMap AllCore.
2   require import WArray64.
3
4   from Jasmin require import JModel JMemory JArray.
5   require import Hacspec.
6   require import Array32 Array16 Array17 Array4.
7
8   require import ZModP.
9
10  type KeyPoly = U8 Array32.t.
11
12  op BLOCKSIZE : usize =
13    16.
14
15  type Block = U8 Array16.t.
16
17  type Tag = u8 Array16.t.
18
19  type FieldCanvas = u8 Array17.t.
20
21  clone import ZModField as ZModField_FieldElement with
22    op p <- 1361129467683753853853498429727072845819
23    rename "zmod" as "FieldElement".
24
```

```

25 op U128Word__from_slice (default: 'a) (x: 'a Seq) (start: int) (len: int) :
    'a Array16.t =
26   Array16.of_list default (take len (drop start x)).
27
28 op ByteSeq_len (x: 'a Seq) : int = size x.
29
30 op FieldElement__from_secret_literal : W128.t -> FieldElement =
    inFieldElement \o W128.to_uint.
31
32 op FieldElement__from_int : int -> FieldElement = inFieldElement.
33
34 op FieldElement__pow2 = FieldElement__from_int \o nat_pow2.
35
36 op ByteSeq_slice (s: 'a Seq) (start: int) (len: int) : 'a Seq =
37   take len (drop start s).
38
39 op FieldElement_to_public_byte_seq_be: FieldElement -> byte_seq = (fun x =>
    nat_to_public_byte_seq_be 17 (asint x)).
40
41 op u128Word__from_seq (x: 'a Seq) : 'a Array16.t = Array16.of_list witness
    x.
42
43 op Tag__from_seq (x: 'a Seq) : 'a Array16.t = Array16.of_list witness x.
44
45 op KeyPoly_slice (s: 'a Array32.t) (start: int) (len: int) : 'a Seq = sub s
    start len.
46
47 op FieldElement__from_literal : W128.t -> FieldElement = inFieldElement \o
    W128.to_uint.
48
49 op ByteSeq_num_chunks (x: 'a Seq) (chunk_size: int) : int =
50   let (q,d) = edivz (size x) chunk_size in
51   q + if (0 < d) then 1 else 0.

```

```

52
53 op ByteSeq_get_chunk (x: 'a Seq) (chunk_size: int) (chunk_num: int) : int *
    'a Seq =
54   let chunk = take chunk_size (drop (chunk_size * chunk_num) x) in
55   (size chunk, chunk).
56
57

```

CÓDIGO B.2: Código do prelúdio do poly1305.

```

1   require import Prelude Hacspec.
2   require import List Int IntDiv CoreMap AllCore.
3   require import WArray64.
4
5   from Jasmin require import JModel JMemory JArray.
6   require import Hacspec.
7   require import Array16 Array64 Array12 Array32 Array4.
8
9
10
11 op state_to_bytes (x_0 : State) : StateBytes =
12   let r_1 = StateBytes__new in
13   let r_1 =
14     foldi (0) (State_len (x_0)) (fun i_2 r_1 =>
15       let bytes_3 = U32_to_be_bytes (x_0.[i_2]) in
16       let r_1 = r_1.[(((i_2) * (4)) <- (bytes_3.[3]))] in
17       let r_1 = r_1.[(((i_2) * (4)) + (1)) <- (bytes_3.[2])] in
18       let r_1 = r_1.[(((i_2) * (4)) + (2)) <- (bytes_3.[1])] in
19       let r_1 = r_1.[(((i_2) * (4)) + (3)) <- (bytes_3.[0])] in
20       r_1)
21   r_1
22 in
23   r_1.
24

```

```

25 op chacha_line
26   (a_4 : StateIdx)
27   (b_5 : StateIdx)
28   (d_6 : StateIdx)
29   (s_7 : usize)
30   (m_8 : State)
31   : State =
32   let state_9 = m_8 in
33   let state_9 = state_9.[a_4 <- ((state_9.[a_4]) + (state_9.[b_5]))] in
34   let state_9 = state_9.[d_6 <- ((state_9.[d_6]) +^ (state_9.[a_4]))] in
35   let state_9 = state_9.[d_6 <- (U32_rotate_left (state_9.[d_6]) (s_7))]
36   in
37   state_9.
38
39 op chacha_quarter_round
40   (a_10 : StateIdx)
41   (b_11 : StateIdx)
42   (c_12 : StateIdx)
43   (d_13 : StateIdx)
44   (state_14 : State)
45   : State =
46   let state_15 = chacha_line (a_10) (b_11) (d_13) (16) (state_14) in
47   let state_16 = chacha_line (c_12) (d_13) (b_11) (12) (state_15) in
48   let state_17 = chacha_line (a_10) (b_11) (d_13) (8) (state_16) in
49   chacha_line (c_12) (d_13) (b_11) (7) (state_17).
50
51 op chacha_double_round (state_18 : State) : State =
52   let state_19 = chacha_quarter_round (0) (4) (8) (12) (state_18) in
53   let state_20 = chacha_quarter_round (1) (5) (9) (13) (state_19) in
54   let state_21 = chacha_quarter_round (2) (6) (10) (14) (state_20) in
55   let state_22 = chacha_quarter_round (3) (7) (11) (15) (state_21) in
56   let state_23 = chacha_quarter_round (0) (5) (10) (15) (state_22) in
57   let state_24 = chacha_quarter_round (1) (6) (11) (12) (state_23) in

```

```

57   let state_25 = chacha_quarter_round (2) (7) (8) (13) (state_24) in
58   chacha_quarter_round (3) (4) (9) (14) (state_25).
59
60 op chacha20_constants_init (_: unit) : U32 Seq =
61   let constants_26 = Seq__new_U32 (secret (u32 8)) (4) in
62   let constants_26 = constants_26.[(0) <- (secret (u32 1634760805))] in
63   let constants_26 = constants_26.[(1) <- (secret (u32 857760878))] in
64   let constants_26 = constants_26.[(2) <- (secret (u32 2036477234))] in
65   let constants_26 = constants_26.[(3) <- (secret (u32 1797285236))] in
66   constants_26.
67
68 op chacha20_key_to_u32s (key_27 : Key) : U32 Seq =
69   let uints_28 = Seq__new_U32 (secret (u32 8)) (8) in
70   let uints_28 =
71     uints_28.[(0) <- (
72       U32_from_le_bytes (
73         U32Word__from_slice_range (secret (u8 8)) (key_27) ((0, 4))))]
74   in
75   let uints_28 =
76     uints_28.[(1) <- (
77       U32_from_le_bytes (
78         U32Word__from_slice_range (secret (u8 8)) (key_27) ((4, 8))))]
79   in
80   let uints_28 =
81     uints_28.[(2) <- (
82       U32_from_le_bytes (
83         U32Word__from_slice_range (secret (u8 8)) (key_27) ((8, 12))))]
84   in
85   let uints_28 =
86     uints_28.[(3) <- (
87       U32_from_le_bytes (
88         U32Word__from_slice_range (secret (u8 8)) (key_27) ((12, 16))))]
89   in

```

```

90   let uints_28 =
91     uints_28.[(4) <- (
92       U32_from_le_bytes (
93         U32Word__from_slice_range (secret (u8 8)) (key_27) ((16, 20)))))]
94   in
95   let uints_28 =
96     uints_28.[(5) <- (
97       U32_from_le_bytes (
98         U32Word__from_slice_range (secret (u8 8)) (key_27) ((20, 24)))))]
99   in
100  let uints_28 =
101    uints_28.[(6) <- (
102      U32_from_le_bytes (
103        U32Word__from_slice_range (secret (u8 8)) (key_27) ((24, 28)))))]
104  in
105  let uints_28 =
106    uints_28.[(7) <- (
107      U32_from_le_bytes (
108        U32Word__from_slice_range (secret (u8 8)) (key_27) ((28, 32)))))]
109  in
110  uints_28.
111
112 op chacha20_iv_to_u32s (iv_29 : IV) : U32 Seq =
113   let uints_30 = Seq__new_U32 (secret (u32 8)) (3) in
114   let uints_30 =
115     uints_30.[(0) <- (
116       U32_from_le_bytes (
117         U32Word__from_slice_range (secret (u8 8)) (iv_29) ((0, 4)))))]
118   in
119   let uints_30 =
120     uints_30.[(1) <- (
121       U32_from_le_bytes (
122         U32Word__from_slice_range (secret (u8 8)) (iv_29) ((4, 8)))))]

```

```

123   in
124   let uints_30 =
125     uints_30.[(2) <- (
126       U32_from_le_bytes (
127         U32Word__from_slice_range (secret (u8 8)) (iv_29) ((8, 12))))]
128   in
129   uints_30.
130
131 op chacha20_ctr_to_seq (ctr_31 : U32) : U32 Seq =
132   let uints_32 = Seq__new_U32 (secret (u32 8)) (1) in
133   let uints_32 = uints_32.[(0) <- (ctr_31)] in
134   uints_32.
135
136 op chacha_block_init (key_33 : Key) (ctr_34 : U32) (iv_35 : IV) : State =
137   State__from_seq (
138     Seq_concat (
139       Seq_concat (
140         Seq_concat (chacha20_constants_init) (chacha20_key_to_u32s (key_33)
141         )) (
142           chacha20_ctr_to_seq (ctr_34))) (chacha20_iv_to_u32s (iv_35))).
143
144 op chacha_block_inner (key_36 : Key) (ctr_37 : U32) (iv_38 : IV) : State =
145   let st_39 = chacha_block_init (key_36) (ctr_37) (iv_38) in
146   let state_40 = st_39 in
147   let state_40 =
148     foldi (0) (10) (fun _i_41 state_40 =>
149       let state_40 = chacha_double_round (state_40) in
150       state_40)
151   in
152
153
154

```

```

155 let state_40 =
156   foldi (0) (16) (fun i_42 state_40 =>
157     let state_40 =
158       state_40.[i_42] <- ((state_40.[i_42]) + (st_39.[i_42]))]
159     in
160     state_40)
161   state_40
162 in
163 state_40.
164
165 op chacha_block (key_43 : Key) (ctr_44 : U32) (iv_45 : IV) : StateBytes =
166   let state_46 = chacha_block_inner (key_43) (ctr_44) (iv_45) in
167   state_to_bytes (state_46).
168
169 op chacha (key_47 : Key) (iv_48 : IV) (m_49 : ByteSeq) : ByteSeq =
170   let ctr_50 = secret (u32 1) in
171   let blocks_out_51 = ByteSeq__new_U8 (secret (u8 8)) (ByteSeq_len (m_49))
172   in
173   let (ctr_50, blocks_out_51) =
174     foldi (0) (ByteSeq_num_chunks (m_49) (64)) (fun i_52 acc =>
175       let (ctr_50, blocks_out_51) =
176         acc
177       in
178       let (block_len_53, msg_block_54) = ByteSeq_get_chunk (m_49) (64) (
179         i_52) in
180       let key_block_55 = chacha_block (key_47) (ctr_50) (iv_48) in
181       let msg_block_padded_56 = StateBytes__new in
182       let msg_block_padded_57 =
183         StateBytes_update_start (msg_block_padded_56) (msg_block_54)
184       in
185       let blocks_out_51 =
186         ByteSeq_set_chunk (blocks_out_51) (64) (i_52) (
187           StateBytes_slice_range (

```

```

186         StateBytes_64_xor W8.(+^) (msg_block_padded_57) (key_block_55))
187     (
188         (0, block_len_53)))
189     in
190     let ctr_50 = (ctr_50) + (secret (u32 1)) in
191     (ctr_50, blocks_out_51))
192     (ctr_50, blocks_out_51)
193 in
194 blocks_out_51.

```

CÓDIGO B.3: Código da tradução do chacha20.

```

1   require import List Int IntDiv CoreMap AllCore.
2   require import WArray64.
3
4   from Jasmin require import JModel JMemory JArray.
5   require import Hacspec.
6   require import Array16 Array64 Array12 Array32 Array4.
7
8
9
10  type State = U32 Array16.t.
11
12  type StateIdx = usize.
13
14  type StateBytes = U8 Array64.t.
15
16  type IV = U8 Array12.t.
17
18  type Key = U8 Array32.t.
19
20  op StateBytes__new = Array64.init (fun _ => W8.zero).
21
22  op State_len (x: 'a Array16.t) : int = 16.

```

```
23
24 op Seq__new_U32 (init: 'a) (size: int) : U32 Seq =
25     mkseq (fun i => W32.of_int 8) size.
26
27 op State__from_seq (x: 'a Seq) : 'a Array16.t = Array16.of_list witness x.
28
29 op ByteSeq_len (x: 'a Seq) : int = size x.
30
31 op ByteSeq_num_chunks (x: 'a Seq) (chunk_size: int) : int =
32     let (q,d) = edivz (size x) chunk_size in
33     q + if (0 < d) then 1 else 0.
34
35 op ByteSeq_get_chunk (x: 'a Seq) (chunk_size: int) (chunk_num: int) : int *
    'a Seq =
36     let chunk = take chunk_size (drop (chunk_size * chunk_num) x) in
37     (size chunk, chunk).
38
39
40
```

CÓDIGO B.4: Código do preludio do chacha20.