Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Miguel Marques Freitas

Implementação Certificada da Componente Criptográfica do Gestor de Passwords KeePass

April 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Miguel Marques Freitas

Implementação Certificada da Componente Criptográfica do Gestor de Passwords KeePass

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Professor José Carlos Bacelar Almeida

April 2022

AGRADECIMENTOS

Ao meu orientador, Prof. Doutor José Carlos Bacelar Almeida, por todo o apoio e disponibilidade que demonstrou durante todo este projeto. Agradeço a paciência e a boa disposição que demonstrou desde o nascer do projeto até à sua conclusão.

Ao professor João Fernando Ferreira do Instituto Superior Técnico de Lisboa, coordenador do projeto *PassCert*, que sempre se mostrou disponível para qualquer situação e acompanhou todo o processo com interesse, boa-disposição e compreensão.

Aos restantes colegas e investigadores do projeto PassCert, que sempre acompanharam a progressão de todo o projeto.

A todos os profissionais de saúde, que também me acompanharam neste caminho e que sempre se mostraram apoiantes e solidários com todos os altos e baixos encontrados.

This work was partially funded by the PassCert project, a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

RESUMO

Com a enorme quantidade de aplicações e sistemas *web* que nos são apresentados existe uma constante preocupação com a nossa segurança e privacidade como utilizadores dos mesmos. Todos nós temos o direito à privacidade dos nossos dados e quando fazemos um registo num novo produto de *software* queremos acreditar que estaremos protegidos de ataques alheios e que, a não ser que a nossa *password* seja descoberta, nenhuma informação nossa vai ser vazada.

Para tal também nos é exigido, consumidores de tecnologia e aplicações, que tomemos uma atitude no sentido de nos protegermos. Uma dessas formas é usar *passwords* seguras e diferentes para cada conta criada. Como isto facilmente se torna impraticável devido à enorme quantidade de contas e, consequentemente *passwords* que é necessário decorar, surgiram os **Gestores de Passwords**. Estes servem para guardar as nossas *passwords* de forma segura e confiável para que sempre que precisemos de uma *password* a irmos buscar de forma simples e rápida.

Assim este projecto visa re-implementar a componente criptográfica do gestor de *passwords KeePass* de forma a garantir os mais altos níveis de confiabilidade e segurança. Para isso, dever-se-á tirar partido das soluções tecnológicas mais recentes para assegurar os referidos níveis de confiabilidade e segurança, como sejam o uso de linguagens de domínio específico para codificação de técnicas criptográficas e sistemas de provas que possam assegurar a respectiva correcção. Para o efeito fazer-se-á uso da linguagem *Jasmin* e do sistema de provas *Easycrypt*.

PALAVRAS-CHAVE Gestor de *Passwords*, Criptografia, KeePass, Jasmin, EasyCrypt

ABSTRACT

With the huge amount of applications and web systems that are presented to us there is a constant concern about our security and privacy as users of them. We all have the right to the privacy of our data and when we register for a new software product we want to believe that we will be protected from outside attacks and that unless our password is discovered, no information about us will be leaked.

This also requires us, consumers of technology and applications, to take action to protect ourselves. One of these ways is to use secure and different passwords for each account created. As this easily becomes impractical due to the huge amount of accounts and consequently passwords that need to be memorized, the Passwords Managers appeared. These serve to store our passwords safely and reliably so that whenever we need a password we get it in a simple and fast way.

So this project aims to re-implement the cryptographic component of KeePass Passwords Manager in order to ensure the highest levels of reliability and security. To this end, the latest technological solutions should be used to ensure these levels of reliability and security, such as the use of domain-specific languages for coding cryptographic techniques and security proof system that can ensure their correctness. For this purpose will be used the Jasmin language and Easycrypt as security proof system .

KEYWORDS Password Managers, Cryptography, KeePass, Jasmin, EasyCrypt.

CONTENTS

LIST OF FIGURES

1

INTRODUÇÃO

1.1 MOTIVAÇÃO

Devido à enorme evolução tecnológica nos últimos tempos, estamos numa era onde existe uma enorme produção de *software* inovador e seguro para responder a todas as necessidades dos consumidores de tecnologia, quer a nível profissional, quer a nível pessoal. É sabido que muitos destes *sites* e aplicações são desenvolvidos para tornar a vida dos utilizadores mais cómoda através de serviços digitais, ajudando-os e simplificando muitos processos que, de outra forma, seriam muito mais complexos e demorados.

Muitos destes serviços implicam registos para se poder aceder aos mesmos, sendo esta uma das maneiras para tentar garantir a segurança e privacidade dos utilizadores. Acompanhado deste elevado crescimento destes produtos de *software* veio também um enorme crescimento do número de *passwords* que um utilizador tem de se lembrar sendo estas *passwords* não só para uso pessoal mas também profissional.

Um estudo feito em 2019 pela *Google Online Security Survey* mostrou que 52% dos utilizadores sondados repetem a mesma *password* em várias aplicações diferentes, sendo este um exemplo muito prático do problema em questão. (1)

O relatório da *2019 Verizon Data Breach Investigations* indica que em 80% da violação e de acesso indevido a dados deve-se a *passwords* fracas, reutilizadas e comprometidas. (2)

Desta necessidade surgiu o conceito de **gestor de *passwords***, que, tal como o nome indica, é um produto que ajuda o utilizador a gerir todas as suas *passwords* de todas as aplicações diferentes. Foram também surgindo vários gestores de *password* com várias funcionalidades e características diferentes, porém com o mesmo objetivo: tornar a experiência do utilizador o mais confiável e segura possível.

1.2 ENQUADRAMENTO

Esta dissertação enquadra-se dentro do projeto **PassCert: Investigação do Impacto de Verificação Formal na Adoção de *Software* para Segurança de *Passwords***. Este tem como objetivo tornar um gestor de *passwords* o mais confiável.

É liderado pelo Instituto Superior Técnico da Universidade de Lisboa em conjunto com a Universidade do Minho e a Universidade da Beira Interior e é apoiado pela *Carnegie Mellon Portugal Program*.

1.3 OBJETIVOS

Este projeto visa então a implementação certificada da componente criptográfica de um gestor de *passwords*, de modo a aumentar os níveis de confiabilidade e segurança. Para tal iremos selecionar um gestor de *passwords* para ser alvo destas alterações.

Numa primeira fase iremos analisar a estrutura e arquitetura deste *software*, de modo a identificar as componentes criptográficas presentes no seu sistema.

Após identificadas todas estas componentes vamos fazer uma abordagem crítica sobre o mesmo para perceber se os algoritmos e soluções que eles apresentam são de facto as mais seguras e as mais confiáveis.

Numa terceira fase iremos começar a re-implementar as componentes identificadas usando as tecnologias que asseguram os níveis de confiabilidade e segurança desejados. Iremos tirar partido do *Jasmin* (3), uma linguagem de domínio específico para codificação de técnicas criptográficas, e o do *EasyCrypt* (4), um demonstrador de teoremas que assegura a correção dos algoritmos implementados.

No final deste projeto pretende-se então tornar um gestor de password mais confiável e seguro. Como muitos dos algoritmos e técnicas criptográficas que irão ser aplicadas neste projeto, são usadas em muitos outros, pretendemos também que este contribuirá para um novo gestor de passwords com o mais alto nível de confiabilidade e segurança.

2

GESTORES DE PASSWORD

O conceito de gestores de *passwords* surgiu de uma necessidade de garantir segurança dos utilizadores e a privacidade dos seus dados de uma forma simples e cómoda para eles.

Este projeto insere-se na implementação certificada da componente criptográfica de um dos muitos gestores de *passwords* disponíveis e por isso é necessário perceber primeiro o que é um gestor de *passwords* e como eles podem ser úteis para um utilizador sem que este comprometa toda a sua informação.

2.1 GESTORES DE PASSWORDS

Como já foi referido anteriormente um gestor de *passwords* é uma aplicação que possibilita ao utilizador guardar as suas *passwords* de forma segura e confiável. Através dele o utilizador poderá criar e utilizar várias contas diferentes em que cada uma delas tem uma *password* considerada forte e segura tendo sempre a garantia que ela estará armazenada no seu gestor para quando precisar dela.

Agora é importante perceber como funciona um gestor de *passwords* de um modo geral. Este tipo de aplicações funcionam como um cofre do utilizador, isto é, os dados são guardados numa base de dados encriptada e apenas podemos aceder a essa informação com uma chave mestre, à escolha do utilizador. É através dessa chave que o utilizador terá acesso a todos os dados e funcionalidades de um gestor de *passwords*, precisando apenas de se lembrar dela.

Com a evolução e o crescimento deste tipo de *software* muitos deles já permitem mais funcionalidades além do cofre. Já é possível, com alguns gestores:

- Fazer preenchimento automático dos campos de *login* : *browsers* e/ou em aplicações *desktop*.
- Gerar *passwords* automáticas e seguras que cumprem os requisitos de *password* exigidas pela plataforma (por exemplo tamanho mínimo e máximo, letras minúsculas e maiúsculas, caracteres especiais).
- Propor guardar palavras-passe automaticamente - alguns gestores conseguem identificar *websites* e aplicações *desktop* que precisam de autenticação e que não tem aquela *password* guardada na base de dados do utilizador.

Estes são só alguns exemplos das funcionalidades que um gestor de *passwords* pode suportar.

Porém a inovação neste tipo de aplicações não é exclusiva a novas funcionalidades. Têm vindo a surgir também novas formas de implementar a chave mestre do utilizador. Agora essas chaves podem não ser apenas

uma palavra-passe. Com a evolução tecnológica e na tentativa de automatizar todos os processos para que a experiência de um utilizador seja cada vez mais simples, fácil e rápida, alguns gestores permitem que um utilizador aceda à sua base de dados através de uma das suas contas. Um exemplo deste tipo de funcionalidade é o gestor de *passwords* da Google. Para podermos aceder às palavras-passe guardadas neste gestor, basta-nos aceder à nossa conta Google através do *browser* e estas estarão disponíveis. Outro exemplo está no gestor de *passwords KeePass* que permite a um utilizador entrar com uma palavra-passe, um ficheiro, ou até com a sua sessão do *Windows*. Também é possível usufruir de mais do que uma maneira de forma a tornar a sua base de dados ainda mais protegida (por exemplo usufruir de uma palavra-chave e um ficheiro, onde o cofre de *passwords* apenas abre quando são introduzidos os dois campos corretamente). (5)

Também o próprio conceito de gestor de *passwords* expandiu para algo mais abrangente. Com o aumento de compras e pagamentos *online*, alguns gestores de *passwords* evoluíram de forma a poderem servir os utilizadores nesse aspeto. Um exemplo deste tipo de serviço está presente no *KeePass2Android* (6), um gestor de *passwords* compatível com o *KeePass* que foi desenhado para *Android*. Nessa aplicação um utilizador consegue guardar, por exemplo, os dados de um cartão de crédito, como o número do cartão, validade e código de segurança (CVV).

Com isto podemos perceber que o conceito de gestor de *password* pode ser expandido e visto como um cofre de informações sensíveis e privadas que permite a um utilizador comum ter uma maior segurança sobre os seus dados e ter sempre acesso a essa informação apenas com alguns cliques. Desta forma garantimos que qualquer acesso a contas de email, contas das finanças, contas bancárias, etc, estão sempre guardadas e protegidas pela palavra-chave única do utilizador.

Estes são apenas alguns exemplos da variedade de funcionalidades e opções que os gestores de *passwords* apresentam nestes dias. Dada a constante preocupação com a segurança e confidencialidade dos utilizadores é importante focar também na garantia que lhes podemos fornecer no que toca à sua segurança, tentando assim fazer implementações mais seguras e tirar partido das melhores tecnologias.

2.2 ALGUNS GESTORES DE PASSWORDS

O conceito de Gestor de *Passwords* tem se tornando cada vez mais popular e cada vez mais usado pelos utilizadores. Naturalmente a oferta deste tipo de produtos tem vindo a aumentar existindo já uma grande variedade de sistemas, cada um com as suas particularidades.

O gestor de *passwords* mais usado a nível mundial é o *Google Chrome Password Manager* (7). Estima-se que seja usado por 2 mil milhões de utilizadores (8), sendo este grátis e integrado no seu próprio *browser*. Apesar de ser construído pela *Google* este é mantido por contribuições em *Open Source*.

À semelhança deste também existe o *Firefox Lockwise* (9), o gestor de *password* da *Firefox* que também é grátis e integrado no seu próprio *browser*. Foi produzido pela *Firefox* e é mantido por eles também.

Porém nem todos estão disponíveis sem qualquer custo para o utilizador. Tendo como exemplo o *1Password* (10) que é um gestor totalmente pago e que em 2019 apresentou 15 milhões de utilizadores chegando a 30 mil entidades de negócio (11) .

Outro exemplo é *LastPass* (12). Este é um *software* com várias funcionalidades disponíveis sem qualquer custo, mas também apresenta funcionalidades *Premium*, que já implica um valor em troca. Este chegou até aos 16,5 milhões de utilizadores e 43 mil entidades de negócio (11).

Outra possibilidade é o nosso gestor de *passwords* alvo: *KeePass*. Este é uma aplicação *desktop*, de acesso gratuito e *Open Source*. Este chegou até aos 20 milhões de utilizadores (11). Neste gestor destaca-se a enorme quantidade de contribuições feitas. Estas incluem novas *frameworks*, *plugins* e extensões para *browsers*. Sendo este um *software* feito para *Windows*, existem também versões compatíveis para os sistemas operativos *MacOS* e *Linux*.

2.3 KEEPASS

O *KeePass* é um gestor de *passwords desktop* e *open source* certificado pela *OSI - Open Source Initiative*. Este é completamente gratuito e é um *software* bastante simples e intuitivo de usar.

Ele permite ao utilizador fazer a gestão das suas *passwords* de forma segura, usufruindo de algumas técnicas criptográficas que mais à frente serão explicitadas e aprofundadas neste documento. O utilizador terá uma base de dados que ficará armazenada na própria máquina. Esta estará encriptada e apenas poderá ser descodificada na aplicação com a palavra-chave correta. Isto permitirá ao utilizador criar várias bases de dados sem que comprometa outras e onde estarão sempre protegidas pela sua chave mestre. Uma vez dentro da base de dados os utilizadores podem usufruir de uma grande quantidade de funcionalidades:

- **Adicionar entradas à base de dados** - Permite a um utilizador adicionar informação que deseja guardar. Aqui ele poderá catalogar a entrada num grupo à sua escolha, adicionar título, *username*, *url*, notas e a *password*. Sempre que ele estiver a inserir uma *password* é-lhe dado um *feedback* da qualidade da *password*. É possível também adicionar *flags* de data de expiração.

- **Gerar *passwords*** - Consoante as indicações do utilizador, o gestor de *passwords* consegue criar *passwords* seguras e aleatórias. Aqui eles podem selecionar várias opções que transmitem à aplicação as regras que têm de cumprir (desde tamanho, caracteres especiais, etc).

- **Adicionar entropia** - De forma a poder gerar valores aleatórios é necessária entropia que influenciará esses mesmos dados a gerar. Para tal o *KeePass* permite a um utilizador adicionar entropia própria e aleatória através de uma caixa de texto para ele introduzir caracteres aleatoriamente e também da posição do cursor numa imagem fornecida.

- **Outras opções** - Nesta categoria encaixam-se opções como *backups* automáticos, eliminar *backups* de entradas quando guardamos a base de dados, apresentar *passwords* expiradas, etc.

Estas são apenas algumas funcionalidades apresentadas ao utilizador. Muitas mais funcionalidades estão escondidas deles e que acontecem sem que o utilizador tenha mão sobre tal. Exemplos destas funcionalidades são a encriptação e desencriptação da base de dados e respetivas entradas, rotinas de segurança, verificação da chave mestre, etc.

O *KeePass* apresenta-nos uma grande variedade de algoritmos e rotinas de segurança. É importante realçar que ele tem duas grandes versões lançadas e disponíveis para instalar: versões **1.X** e **2.X** . Ambas são versões estáveis e funcionais de um gestor de *passwords*, porém existem algumas diferenças a nível de algoritmos e rotinas de segurança. Naturalmente as versões 2.X têm mais procedimentos de segurança e atualizaram alguns algoritmos em comparação com as versões 1.X, tomando como exemplo :

- *Novas funcionalidades* - operações funcionais que não influenciam a segurança do *software* mas ajudam os utilizadores a ter uma melhor experiência.
- *Novos algoritmos de encriptação*
- *Novos tipos de chaves mestre* - são possíveis chaves como cartões NFC, certificados, entre outros
- *Mais adaptadas e maior quantidade de plugins*

No capítulo 3 vamos fazer uma análise mais pormenorizada às técnicas criptográficas e de segurança usadas neste gestor de *password* onde vamos listar algoritmos de encriptação, algoritmos de derivação de chaves, processos de garantia de autenticidade e integridade e mais. Além desta listagem vamos também tentar perceber se estes algoritmos são aceites e confiáveis.

3

ESTADO DA ARTE

Um gestor de *passwords* tem de ter uma forte componente criptográfica para proteger os dados dos utilizadores. Se analisarmos o conceito de um gestor vemos que é intrínseco a necessidade de ele ser seguro, confiável e íntegro para dar aos utilizadores aquilo que ele diz ser e fazer. Com isto é importante que os algoritmos criptográficos e operações de segurança sejam de facto atualizadas e que haja garantias que sejam seguras.

Neste capítulo vamos apresentar as características de segurança que o *KeePass* nos apresenta e os algoritmos que utiliza.

3.1 TÉCNICAS CRIPTOGRÁFICAS RELEVANTES

Antes de analisarmos a componente criptográfica do nosso gestor de *passwords* é sensato recordar alguns conceitos de criptografia. Nesta secção vamos abordar algumas técnicas básicas e relevantes na criptografia de forma a mais tarde, neste documento, percebermos a relevância do levantamento feito à componente criptográfica do *KeePass*.

Assim vamos recordar os conceitos de cifra, funções de *hash* criptográficas, funções de derivação de chaves, *MACs* e funções pseudo-aleatórias.

3.1.1 *Cifras*

As cifras são técnicas criptográficas para esconder um texto tornando-o privado para apenas aqueles que têm a chave.

O exemplo mais simples desta técnica pode ser dado por:

$$criptograma = Encriptar(chave\_encriptacao, mensagem)$$

No exemplo em cima vemos que o criptograma é obtido através de um algoritmo de encriptação e recebe como argumento a chave e a mensagem a esconder.

Para desencriptar o processo será o inverso, isto é, pegamos na chave de desencriptação e aplicamos o algoritmo de desencriptação ao criptograma, obtendo assim a mensagem:

$$mensagem = Desencriptar(chave\_desencriptacao, criptograma)$$

Esta é a demonstração genérica de como as cifras são utilizadas havendo muitas diferenças entre os diferentes algoritmos existentes.

Existem algoritmos que usam chaves simétricas - a chave de encriptar é a mesma para decifrar o criptograma - ou chaves assimétricas - cada interveniente tem um conjunto de chaves, uma pública e privada, onde o emissor encripta a sua mensagem com a chave pública do recetor e este desencripta-a com a sua chave privada:

$$Emissor : criptograma = Encriptar(chave\_publica\_Recetor, mensagem)$$

$$Recetor : mensagem = Desencriptar(chave\_privada\_Recetor, criptograma)$$

Também temos diferenças quanto à abordagem do próprio algoritmos à mensagem que recebe, podendo esta ser uma *stream cipher* - onde cada dígito da mensagem é processado - ou uma *block cipher* - a mensagem é partida em blocos de tamanho *n* e o criptograma é formado pela junção do processamento de cada bloco.

Existem também alguns modos de como as cifras atuam, podendo destacar três dos mais conhecidos:

- **ECB** - *Electronic Codebook*

  Parte uma mensagem em blocos e encripta cada bloco individualmente:

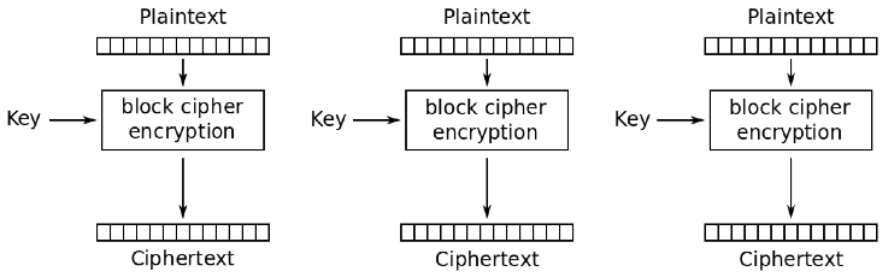

Figure 1: Modo ECB - Electronic Codebook

- **CBC** - *Cipher Block Chainning*

  Parte uma mensagem em blocos e encripta cada bloco juntamente com o criptograma do bloco anterior. Ao bloco da mensagem é aplicado um XOR com o criptograma do bloco anterior e depois é encriptado. No primeiro bloco é usado um vetor de inicialização (IV):

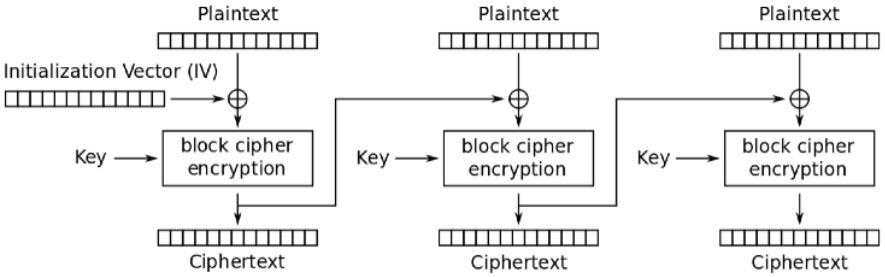

Figure 2: Modo CBC - Cipher Block Chainning

- **CFB** - *Cipher Feedback*

  Tal como o anterior este modo usa o resultado da encriptação do bloco anterior para encriptar o bloco atual. A diferença é que neste encriptamos o criptograma anterior e só depois aplicamos o XOR à mensagem. Para a primeira iteração é também usado um vetor de inicialização:

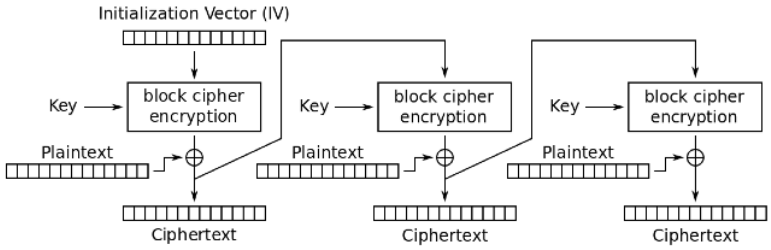

Figure 3: Modo CFB - Cipher Feedback

Estes são apenas alguns exemplos de vários tipos e modos de cifra atuarem sobre uma mensagem, existindo muito outros. Das várias cifras podemos destacar:

- **AES** - Aprovada pela NSA (*National Security Agency*), nos Estados Unidos, como algoritmo confiável e seguro para guardar informações ultrassecretas. É também o algoritmo *standard* adoptado pelo *U.S. federal government*. Este algoritmo é um subconjunto de cifra de blocos da família *Rijndael* e é um algoritmo de chave simétrica.
- **ChaCha20** - É uma variante do algoritmo **Salsa20**. É uma *stream cipher* e é geralmente mais rápida que a **AES**.
- **TwoFish** - Cifra por blocos que foi um dos cinco finalistas no concurso para o novo *AES*.

3.1.2 *Funções de Hash*

Funções de *hash* são funções que recebem argumentos de tamanho arbitrário e produzem um resultado de tamanho fixo. Muitas são as aplicações que estas podem ter no mundo da tecnologia, sendo usadas em

estruturas de dados, como tabelas de *hash*, e muitos outros. Pela sua definição conseguimos perceber que o domínio de uma função de *hash* é infinito enquanto que o seu contra-domínio é limitado e finito. Por isto percebemos também que dois *inputs* diferentes podem ter o mesmo *output*. Uma função de *hash* pode ser muito útil em diversas situações inclusive na criptografia, porém é necessário propriedades extras de forma a garantir que esta seja segura, surgindo assim as funções de *hash* criptográficas.

Estas além de comprimirem os argumentos recebidos para um tamanho fixo também têm as seguintes propriedades:

- **Modificar uma mensagem *m* também modifica o seu *hash* correspondente** - usufruem do "efeito de avalanche", isto é, uma alteração num bit irá ter efeito em todos os bits seguintes.
- **É computacionalmente impossível encontrar uma mensagem *m* através de uma *hash*** - o processo de transformar uma mensagem numa *hash* é bastante simples e rápido, porém o processo inverso é muito mais complexo e difícil, sendo por isso uma função uni-direcional.
- **É computacionalmente impossível encontrar duas mensagens diferentes com a mesma *hash*** - desta forma a probabilidade de haver colisões é tão baixa que deixa de ser tornar viável encontrar duas mensagens diferentes com o mesmo resultado.

Todas estas propriedades foram implementadas de forma a garantir que não deverá ser possível reverter o resultado de uma função de *hash* de forma eficiente.

Para este tipo de funções podemos destacar os algoritmos:

- **MD5** - Função criptográfica que foi muito popular até encontrarem vulnerabilidades. Pode ser utilizada para garantir a integridade de dados quando a corrupção dos mesmos é não intencional.
- **SHA-1** - Função de *hash* criptográfica desenvolvida pela *National Security Agency* e aprovada pela *NIST*. Tal como a **MD5** foi muito utilizada até terem descoberto algumas fraquezas teóricas, sendo quebrada em 2017.
- **SHA-2** - Função de *hash* que surgiu no seguimento da **SHA-1**.
- **Keccak** - Função que em 2012 venceu a *NIST hash function competition*. Esta competição foi feita para encontrar um sucessor para o algoritmo **SHA-1** depois de se terem encontrado as suas vulnerabilidades teóricas. Este algoritmo ficou denominado por **SHA-3** depois de ter vencido a competição.

3.1.3 *Funções de Derivação de Chaves - KDF's*

Uma *Key Derivation Function* é uma função que, tal como o nome indica, serve para derivar uma chave.

Este tipo de funções recebem *inputs*, como palavras-chave, e transformam esse mesmo *input* de forma a torná-los adequados para operações criptográficas, tirando partido de funções pseudo-aleatórias.

Podemos identificar dois principais usos para este tipo de funções (13):

- **Derivar chaves criptográficas** - recebem uma palavra-chave por *input* do utilizador e transforma-la de modo a ser usada em algum algoritmo criptográfico.
- **Armazenar *passwords*** - de forma a armazenar *passwords* de forma segura pretende-se que o algoritmo seja pesado. Assim enquanto um utilizador que saiba a sua *password* apenas tem de executar o processo uma vez e compara com o valor armazenado, um atacante terá de executar muitas vezes, sendo assim um processo demorado.

Neste processo é usado um *salt* aleatório de forma a acrescentar alguma aleatoriedade. Também tem em conta o número de iterações que se deseja fazer sobre o *input*.

Ataques de dicionário não são possíveis de se evitar, por isso resta-nos tornar essas tentativas o mais difíceis e complexas possíveis. Este tipo de ataques baseiam-se em tentar adivinhar a chave de desencriptação à força bruta. Uma técnica para facilitar este trabalho seria usar palavras de um dicionário que fosse familiar ao utilizador, como nomes do próprio ou de familiares, números que lhe fossem especiais, atividades de lazer do utilizador, entre outros. Estes mecanismos destinam-se a evitar que o adversário tire partido de chaves e ataques de dicionário e força bruta.

Existem muitos algoritmos que se encaixam neste tipo de funções destacando assim:

- **AES-KDF** - Função de derivação de chaves baseado no algoritmo no *AES*.
- **Argon2** - Algoritmo desenhado para ter uma maior resistência a ataques de dicionários mais modernos aumentando a dificuldade e o esforço a nível de *CPU* mesmo que estes usem plataformas de paralelismo bastante evoluídas. Foi também vencedor do *Password Hashing Competition*.
- **HKDF** - Função de derivação de chaves baseado em *hash macs*. Permite transformar chaves partilhadas entre intervenientes em chaves de encriptação e verificação de autenticidade e integridade.
- **PBKDF2** - Algoritmo desenhado para derivar *inputs* em chaves de encriptação.
- **Scrypt** - Tal como o *Argon2* este algoritmo foi desenhado a pensar em ataques de dicionário. Idealmente utilizada para armazenar *passwords*.

3.1.4 *MACs - Message Authentication codes*

Message Authentication Codes são algoritmos que produzem um conjunto de *bits* que permitem a verifação da integridade e autenticidade de um criptograma.

Estas funções recebem uma chave e a mensagem que querem autenticar como *inputs* e devolvem um conjunto de *bits*. Esses mesmo *bits* são denominados de *MAC* (também conhecidos por *tag*).

Este processo garante a integridade e a autenticidade de uma mensagem pois permite ao verificador ver se a mensagem foi alterada ou corrompida. Os algoritmos *MAC* têm um algoritmo que com a chave de verificação, a *tag* e a mensagem a verificar conseguem verificar se de facto a mensagem é autêntica.

Uma forma de implementar um *MAC* é recorrendo as funções de *hash* criptográficas, obtendo assim as *hmacs* - *hash-based message authentication code*. Qualquer função de *hash* pode ser usada no cálculo deste *hmac* e também este permite a verificação da integridade dos dados e a autenticidade de uma mensagem. Como exemplo deste tipo de funções temos o **HMAC-SHA-256** onde o processo de gerar mensagens de autenticação é obtido pela função de *hash* **SHA-256**.

Existem muitas formas de combinar MACs com cifras, por forma a garantir simultaneamente autenticidade e confidencialidade, sendo destacados três esquemas: *Encrypt-and-MAC*, *MAC-then-Encrypt* e *Encrypt-then-MAC*.

Encrypt-and-MAC

Caracteriza-se pelos processos de encriptação e de *MAC* serem aplicados ao texto limpo:

$$criptograma = Encriptar(Chave_{cifra}, texto\_limpo)$$

$$tag = MAC(Chave_{mac}, texto\_limpo)$$

O processo de verificação da integridade do texto limpo passará por desencriptar o criptograma e depois aplicar o *MAC* ao texto desencriptado e comparar com a *tag* recebida.

Este esquema apenas permite a verificação da integridade do texto desencriptado mas não é uma escolha segura. Como o criptograma não está protegido este pode ser atacado. Além disso este esquema pode revelar informações da mensagem na própria *tag*. (14)

MAC-then-Encrypt

Este esquema aplica a função *MAC* ao texto limpo e depois encripta tanto a *tag* resultante dessa operação como o texto limpo:

$$tag = MAC(Chave_{mac}, texto\_limpo)$$

$$criptograma = Encriptar(Chave_{cifra}, tag + texto\_limpo)$$

Tal como o anterior, apenas garante integridade do texto limpo e passará por desencriptar o criptograma. Depois obtemos a *tag* e com o texto já desencriptado podemos verificar a sua integridade. Esta é ligeiramente melhor que a anterior visto que não exibimos nenhuma informação sobre o texto limpo (pois a *tag* também está encriptada).

Encrypt-then-MAC

Por último temos o esquema que se caracteriza por garantir a integridade do criptograma. Este irá encriptar uma mensagem e só depois irá aplicar o algoritmo *MAC* sobre o criptograma resultante:

$$criptograma = Encriptar(Chave_{cifra}, texto\_limpo)$$

$$tag = MAC(Chave_{mac}, criptograma)$$

Este esquema é o esquema considerado mais seguro pois a verificação da integridade e autenticidade é aplicado ao criptograma, antes de qualquer operação de desencriptação. Desta forma o verificador protege-se de qualquer ataque de implementação que poderiam vir disfarçados no criptograma.

3.1.5 *Pseudo-Random Functions*

Funções pseudo-aleatórias são funções que, apesar de serem um processo inteiramente determinístico, produzem sequências que apresentam aleatoriedade estatística.

Este tipo de funções são usadas em algumas técnicas criptográficas de forma a acrescentar aleatoriedade ao processo de forma a tentar esconder parâmetros usados na encriptação da mensagem, sendo frequentemente usadas na derivação de chaves.

3.2 KEEPASS

O *KeePass* será o gestor de *passwords* escolhido para ser alvo deste projeto. Antes de começarmos a implementar a nova versão da sua componente criptográfica foi feito um levantamento das suas características quer a nível de segurança, quer a nível de algoritmos criptográficos.

3.2.1 *Segurança*

Técnicas e procedimentos de segurança são aliados da criptografia, pois criar um *software* seguro não se limita a aplicar algoritmos de encriptação. Muitas são as rotinas a implementar numa aplicação para que esta seja considerada segura. Uma característica clara do *Keepass*, como um gestor de *passwords*, é a garantia de este ser seguro para o cliente, de forma a garantir a confiabilidade e privacidade desejadas.

Base de dados

Para proteger todos os dados do utilizador o *Keepass* encripta todos as informações a serem guardadas, em vez de proteger apenas as informações mais críticas, como a *password*. Este procedimento garante ao utilizador uma maior segurança dos seus dados, visto que nenhuma informação é vista por terceiros sem que estes saibam a chave mestre do utilizador. Quando o tipo de dados guardados requer vários campos, como por exemplo guardar dados de um cartão de crédito, o facto da base de dados ser toda cifrada é um ponto muito favorável para o utilizador.

Além de cifrar toda a base de dados, sempre que uma base de dados é guardada é gerado um vetor de inicialização aleatório, de forma a acrescentar aleatoriedade no processo de encriptação dos dados.

Chaves

A chave do utilizador é o maior segredo a ser guardado pelo utilizador, pois será essa chave que irá aceder à informação toda do seu cofre pessoal.

Por isso é importante nos protegermos contra tentativas de adivinhar a chave do utilizador. Quanto mais complexo for o processo de acertar numa chave por "força bruta", mais seguro estará o utilizador. Para tal poderemos recorrer a alguns procedimentos.

Uma das formas de o fazermos é permitir ao utilizador introduzir uma chave de um tamanho a gosto. Dessa maneira acrescentamos dificuldade a um possível atacante, visto que ele não terá acesso ao tamanho da *password*.

À chave do utilizador é aplicada uma *Key Derivation Function* de forma a criar uma nova chave para o processo de encriptação dos dados.

Proteção contra ataques de dicionário

Como já foi dito no ponto acima, são aplicados algoritmos de derivação de chaves sob a chave mestre do utilizador de forma a dificultar possíveis ataques de dicionário.

No que toca ao número de iterações, é possível ao utilizador aumentar e diminuir como ele quiser, assim como escolher o algoritmo que prefere (esta escolha apenas é permitida nas versões **2.X** visto que estas suportam mais que um algoritmo como iremos ver mais à frente neste documento). Outra opção que o utilizador pode selecionar é a opção *1 Second Delay*. Esta opção diz ao *KeePass* para iterar o número de vezes que conseguir durante um segundo.

Geração aleatória de números

Para executar funções criptográficas seguras é necessário adicionar aleatoriedade em alguns momentos. Alguns algoritmos incorporam operações de gerar números aleatórios, como as funções de derivação de chaves quando precisam de um valor, o *salt*, ou então algoritmos de encriptação que precisam de um vetor de inicialização também aleatório. Para tal o *KeePass* usa uma função *PRNG* (*Pseudo-Random Number Generator*) criptograficamente segura, que é baseada nos algoritmos *SHA-256*/*SHA-512* e *ChaCha20*. Para esta função é necessária uma entropia inicial.

Essa entropia tem de ser alimentada também por fontes aleatórias e por isso o nosso gestor de *passwords* combina várias fontes diferentes de forma a complicar e a aumentar variáveis para obter um resultado que seja difícil de prever. Alguns exemplos dessas fontes de entropia são números aleatórios gerados pela sistema criptográfico, data e hora atuais, posição do cursor, versão do sistema operativo, variáveis de ambiente e muitos mais.

Proteção contra falhas de memória de processo

Entende-se por falha de memória de processo quando ocorre uma falha no sistema e este envia um processo diretamente para o disco. Se nada for feito no intuito de se proteger perante esta falha, significa que o processo do *KeePass* que estava a correr iria diretamente para o disco e armazenaria-se o estado imediatamente anterior à falha. Isto levaria a uma revelação dos dados que estavam expostos nesse mesmo processo, e estes estariam sem qualquer proteção.

Para evitar essas falhas as informações mais sensíveis, que neste caso são as *passwords*, são logo encriptadas enquanto o *software* está a correr. Desta forma, mesmo que o processo falhe e seja enviado para o disco, reduz consideravelmente a viabilidade dos ataques do adversário.

Introdução da chave mestre num ambiente seguro

Uma das técnicas de tentar descobrir uma *password* é fazer análise aos *logs* (registo das operações) do utilizador. Por vezes os utilizadores enganam-se a introduzir a *password* e muitos *logs* expõe esses dados, sendo por isso uma fonte bastante útil de informação para um atacante, pois conseguem ver tentativas e pode conter pistas e indícios das *passwords*. Um exemplo claro disso é quando um utilizador tem de introduzir credenciais de autenticação e além de introduzir o seu *username* acaba por introduzir também a *password* na mesma caixa de texto. Estes tipos de erros comuns levam a uma exposição das credenciais e coloca o utilizador em perigo.

Existe também *software* que conseguem gravar os *inputs* do utilizador quando estes estão no seu ambiente de trabalho normal, os *keyloggers*. Através deste tipo de *software* um atacante poderá conseguir distinguir alguns inputs de *passwords*.

O *KeePass* previne essas situações pois apresenta um ambiente seguro para introduzir a *password* que não é alcançado por quase nenhum *keylogger*.

Testes automáticos

As funcionalidades básicas de segurança do *KeePass* são as funções de encriptação/desencriptação. De modo a garantir que esses algoritmos estão funcionais sempre que o *software* é iniciado ele executa testes aos algoritmos segundo um vetor de testes do mesmo. No caso de algum algoritmo não passar em algum teste é apresentado um aviso ao utilizador.

Bloquear ambiente de trabalho do KeePass

É possível o utilizador bloquear o seu ambiente de trabalho atual (*workspace*). Quando ele o bloqueia fecha a base de dados, encriptando todos os dados, ficando apenas com o caminho para o ficheiro e alguns parâmetros de visualização. Isto fornece segurança máxima, visto que desbloquear o *workspace* é tão difícil como abrir a base de dados quando inicializamos o *software*.

Além disso previne perda de dados e no caso do computador ter uma falha que o leva a reiniciar os processos, se a base de dados estiver bloqueada não irá ser afetada por esta falha.

Importação e Exportação de dados

É possível ao utilizador fazer a importação/exportação de dados de/para vários formatos diferentes. Isto permite ao utilizador fazer cópias de segurança ou até exportar os dados para importar e guardar noutra máquina.

São muitos os formatos possíveis para estas operações desde *.txt*, *html*, *csv* que torna o ficheiro completamente compatível com outros gestores de *passwords* e também aplicações como o *Microsoft Excel* e o *LibreOffice Calc*, ou ainda *xml* que pode facilmente integrar e ser usado noutras aplicações.

3.2.2 *Algoritmos Criptográficos*

Além das operações e rotinas de segurança, o *KeePass* apresenta também algoritmos criptográficos seguros que são mundialmente aceites como tal.

Por outras palavras, este gestor usufrui de técnicas reconhecidas e certificadas que garantem ao utilizador a privacidade desejada. Além disso vemos que muitos desses algoritmos são usados em muitos outros *software*, desde produtos para bancos e finanças, ou até para intuito militar.

Base de dados

Como sabemos este *software* encripta toda a base de dados, incluindo *usernames*, *passwords*, notas, etc, sendo possível usar três algoritmos diferentes.

Para versões **1.X** podemos optar entre os algoritmos:

- **AES/Rijndael** - Cifra por blocos, aprovada pela *NSA* (*National Security Agency*) e adoptado pelo *U.S. federal government*.
- **Twofish** - Cifra por blocos, algoritmo que foi outro dos cinco algoritmos finalistas *AES*.

Para versões **2.X** temos disponíveis:

- **AES/Rijndael** - Cifra por blocos foi aprovado pela *NSA* e adotado pelo *U.S. federal government*.
- **ChaCha20** - Cifra sequencial sucessora do algoritmo *Salsa20*.
- **Outros** - através de *plugins* são possíveis usar algoritmos como *Twofish*, *Serpent* e *GOST*, entre outros.

Em todos os casos são usadas chaves de 256 *bits*.

Nos algoritmos de cifras por blocos são usadas operações no modo **CBC** - *Cipher Block Chaining*.

Além disso sempre que uma base de dados é guardada é gerado aleatoriamente um vetor de inicialização, **IV**.

Integridade e autenticidade

Na encriptação da base de dados podemos encontrar algoritmos que garantem integridade e autenticidade dos dados guardados. Estes algoritmos permitem ao *software* verificar se os dados foram comprometidos ou corrompidos.

Para tal podemos ver que as versões **1.X** usam o algoritmo **SHA-256** para produzirem uma *hash* sobre o texto limpo.

Nas versões **2.X** utiliza-se o algoritmo **HMAC-SHA-256** para produzir um *MAC* dos dados encriptados.

Hash e derivação da chave

Uma das características do *KeePass* é que este permite ao utilizador utilizar uma chave mestre composta. Essa chave pode ser constituída por uma palavra-chave em conjunto com um ficheiro (*keyfile*) , chaves provenientes de *plugins* ou até um utilizador *Windows*. Isto implica que o tamanho da chave mestre possa ter vários tamanhos diferentes, sendo por isso aplicada o algoritmo **SHA-256** sobre a chave composta, de modo a garantir que a chave mestre tenha sempre 256 bits.

A esta nova chave-mestre gerada será aplicada também um algoritmo de derivação de chave com um *salt* aleatório, para que esta seja usada nos algoritmos de encriptação. Esta nova chave derivada vai encriptar e desencriptar os dados da base de dados.

Para este efeito podemos ver o algoritmo **AES-KDF** na versão **1.X** do *KeePass*. É possível ao utilizador mudar o número de iterações feitas sobre a sua chave, porém o tempo de guardar ou carregar a sua base de dados é diretamente proporcional ao número de iterações.

Já as versões **2.X** do *KeePass* suportam os algoritmos **AES-KDF** e **Argon2** em três variantes Argon2d, Argon2id e Argon2i. Este algoritmo Argon2 foi o vencedor do *Password Hashing Competition*.

Na seguinte imagem podemos ver todo o processo desde a chave mestre do utilizador até obtermos a chave de encriptação.

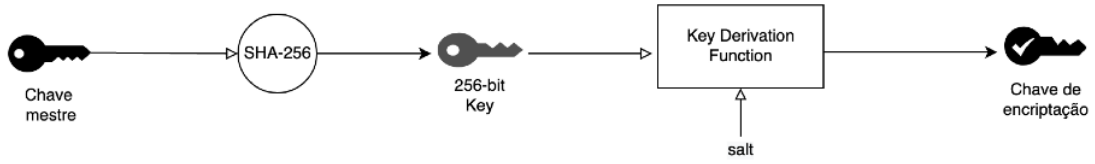

Figure 4: Processo de compressão e derivação da chave mestre do utilizador

Em suma

Sumarizando temos presentes os seguintos algoritmos:

- Encriptação/Desencriptação
  - **Twofish** - Versões 1.X

  - **AES/Rijndael** - Versões 1.X e 2.X
  - **ChaCha20** - Versões 2.X
- Integridade e Autenticidade
  - **SHA-256** *sobre o texto limpo* - Versões 1.X
  - **HMAC-SHA-256** *no esquema Encrypt-then-MAC* - Versões 2.X
- *Hash* e derivação da chave
  - *Hash* da chave mestre do utilizador:
    - **SHA-256** - Versões 1.X e 2.X
  - Derivação das chaves:
    - **AES-KDF** - Versões 1.X e 2.X
    - **Argon2** *nas variantes Argon2d, Argon2id e Argon2i* - Versões 2.X

3.3 FERRAMENTAS

Depois da pesquisa e análise à componente criptográfica dos gestores de *passwords* veio também uma pequena pesquisa sobre as ferramentas que nos irão acompanhar ao longo deste projeto. Durante a formulação deste projeto do **PassCert** foram escolhidas estas ferramentas devido à envolvência que os docentes e os participantes tinham com a mesma.

3.3.1 *Jasmin*

O *Jasmin* é uma *framework* para desenvolver software criptográficos de alto desempenho e fiabilidade. Esta *framework* foi construída em torno da sua própria linguagem e compilador. A linguagem foi desenhada de forma a realçar a portabilidade dos seus programas e facilitar os processos de verificação dos mesmos. Já o seu compilador foi construído de modo a conseguir alcançar previsibilidade e eficiência do código compilado e transformado em código *Assembly*, devido às suas parecenças, estando limitado a plataformas de 64 bits. Este compilador está formalmente verificado pelo *Coq proof assistant*, um sistema de gestão de verificações formais (15).

Durante a compilação de um programa em *Jasmin* a semântica da linguagem é respeitada e mantida, isto é, as instruções do código *Assembly* gerado são fidedignas ao código em *Jasmin*.

Durante a compilação também é feita outra verificação que testa propriedades de segurança como *memory safety* (impedir acessos a zonas de memória às quais não temos acesso) e operações matemáticas impossíveis (por exemplo divisões por zero).

Esta *framework* também nos permite extrair os programas para programas correspondentes em *EasyCrypt* de modo a podemos provar correção e segurança dos algoritmos. Com esta funcionalidade permite-nos também verificar a propriedade de *constant-time*, que iremos explicar mais tarde neste documento.

Sintaxe Jasmin

Jasmin foi a *framework* escolhida para implementar os algoritmos. Esta *framework* permite cumprir alguns requisitos de um bom *software* criptográfico como **eficiência**, **proteção contra *side-channel attacks***, e **correção funcional**.

Esta caracteriza-se por ser uma linguagem de baixo nível, muito semelhante à linguagem C, porém permite a construção de código criptográfico de alto desempenho. São possíveis operações como limitar *overhead* de funções, alocar registos para variáveis e ainda invocar instruções *Assembly* diretamente no código, que permite obter um melhor desempenho dos algoritmos.

Vamos agora fazer uma análise à sintaxe base para podermos perceber como esta linguagem funciona.

Tipo de dados

Os números inteiros são um tipo de dados indispensáveis para a programação. Assim o *Jasmin* permite representar inteiros sem sinal através do prefixo **u** seguido do número de *bits* que o mesmo ocupa.

Um número inteiro também pode ser representado pelo prefixo `inline int` que é usado no *unrool* de *loops*. Esta opção permite que esse *unroll* seja feito em tempo de compilação e apesar de aumentar o tamanho do código resultante leva a uma poupança de instruções de cálculo e controlo:

```
u8 a; //inteiro com 8 bits
u16 b; //inteiro com 16 bits
u32 c; //inteiro com 32 bits
u64 d; //inteiro com 64 bits
inline int i;
```

A representação do `inline int` também pode ser usada como *flags* do CPU alteradas em operações *Assembly*. Essas mesmo *flags* também podem ser representadas com o tipo `bool` que assumem os valores normais de *True* (quando o inteiro é 1) ou *False* (quando o inteiro é 0).
Para declarar *arrays* usamos a nomenclatura `TIPO[TAMANHO] Nome`. Esta nomenclatura diz que o *array* `Nome` tem `TAMANHO` elementos do tipo `TIPO`. Um exemplo mais concreto dessa declaração seria:

```
u64[10] arr;
```

Além da representação de variáveis é possível também aplicar modificadores, como é o caso do *inline*. Temos também os modificadores *stack*, que delega variável para a *stack*, e *reg* quando queremos que uma variável seja mantida em registo:

```
inline int i; //inteiro usado no loop onde o compilador vai fazer
//loop unrolling
stack u64[10] arr; //array de 10 inteiros de 64bits armazenado na stack
reg u64 b; //inteiro de 64bits guardado nos registos
```

Estas modificações permitem melhorar a *performance* do programa com indicações ao compilador. O modificador *reg* permite manter o seu valor em registo reduzindo o *overhead* nas operações onde esta variável é usada. Já o modificador *stack* são bastante úteis quando temos *arrays* de grandes dimensões visto que não é viável o armazenamento deles em registos devido à limitação do número registos impostos pelo *hardware*.

Loops e Condições

O *Jasmin* tem uma sintaxe muito idêntica à linguagem C no que diz respeito a condições e *loops*.
As estruturas de condições **if/else** mantém-se tal e qual ao C.

```
if(condition1) {
   /*
   INSTRUÇÕES
   */
}
else {
    /*
    OUTRAS INSTRUÇÕES
    */
}
```

Já os *loops* permite também **for** e **while** mas têm uma melhoria significativa no que toca à *performance*. O já referido *loop unrolling* permite que o compilador, em tempo de compilação, calcule os valores que as variáveis do *loop* vão assumir nas várias iterações.

```
inline int i, j;
i = 0;
while(i < 10) {
    /*
    INSTRUÇÕES
    */
    i+= 1;
}
```

```
for(j = 0; j < 10; j +=1){
    /*
    INSTRUÇÕES
    */
}
```

Funções

Por último temos a sintaxe das funções. As funções de *Jasmin* seguem o formato de *keyword* **fn**, nome da função, parâmetros (uma lista composta por `TIPO NOME`), **->** e tipo retornado:

```
fn NOME_DA_FUNÇÃO(TIPO_PARÂMETRO NOME_PARÂMETRO, ...)
 -> TIPO_RETORNADO {
    /*
    INSTRUÇÕES
    */
}
```

Tanto os parâmetros como o tipo do elemento a ser retornado podem ser vazios.

O *inlining* de uma função é uma instrução de otimização da compilação que diz ao compilador para substituir a chamada de uma função pelo corpo dessa mesma função, permitindo poupar um conjunto de instruções implícitas de fazer chamadas. Para podermos indicar ao compilador que queremos essa otimização temos de a utilizar como prefixo de uma função. No seguinte excerto vamos ver a implementação da função *shift* em *Jasmin*:

```
inline fn shift(reg u128 x, inline int count) -> reg u128{
    reg u128 r;
    r = #VPSLL_4u32(x,count);
    return r;
}
```

Além do prefixo *inline* é possível aplicar o *export* que é usado quando queremos que uma função possa ser utilizada noutro programa ou módulo:

```
export fn sum(reg u64 x, reg u64 y) -> reg u64{
    reg u64 r;
    r = x + y;
    return r;
}
```

Esta função agora pode ser chamada não só noutro programa em *Jasmin*, mas também noutro programa de outra linguagem como por exemplo um programa em C:

```
extern int sum (uint64_t x, uint64_t y);

int r = sum(5,10);
```

Uma particularidade da sintaxe de uma função é que também o corpo da função tem uma estrutura definida que passa por primeiro fazer a declaração de todas as variáveis necessárias, seguidas das instruções e terminando com o retorno de uma variável (quando aplicável).

```
fn exemplo(reg u128 x, reg u128 y) -> reg u128{
    //Declarar variáveis
    reg u128 r, z;
    inline int count;

    //Instruções
    z = x * y ;
    count = x - y ;
    r = #VPSLL_4u32(z,count);

    //Retorno
    return r;
}
```

3.3.2 *EasyCrypt*

EasyCrypt é um conjunto de ferramentas cujo principal objetivo é a construção e verificação de provas baseadas em jogos criptográficos. Por outras palavras esta ferramenta permite-nos fazer provas de segurança sobre primitivas criptográficas.

Assim com este assistente de provas poderemos especificar sintaxes e modelos de segurança para protocolos criptográficos, especificar soluções criptográficas (quando estas são necessárias) assim como protocolos criptográficos concretos. Permite-nos também provar a correção e segurança desses mesmo protocolos.

Esta ferramenta está muitas vezes associada à *framework* do *Jasmin* sendo bastante comum vê-las em conjunto para provar correção e segurança dos seus programas também.

3.3.3 *Outras Ferramentas*

Apesar das ferramentas selecionadas serem as *frameworks Jasmin* e *EasyCrypt* existem muitas outras pelas quais poderíamos ter optado. Não obstante que o projeto teria estas ferramentas como pressupostos foi feito um pequeno levantamento de outras ferramentas que poderiam também servir para este efeito.

Como exemplo temos as ferramentas ***HACL*, Vale* e *EverCrypt***. *HACL** fornece implementações eficientes de primitivas criptográficas em *C* puro, enquanto que a *framework Vale* consegue providenciar código *Assembly* otimizado para operações onde a *performance* é crítica. O *EverCrypt* é uma API de alto nível, na linguagem *C*, que seleciona automáticamente a melhor implementação, entre *HACL** e *Vale*, para a máquina onde o algoritmo está a ser corrido.

Uma das vantagens de utilizar este conjunto de ferramentas seria a facilidade de adaptação ao gestor de *passwords* visto que já contém uma vasta lista de algoritmos criptográficos implementados.

Outra ferramenta seria o demonstrador de teoremas **CertiCrypt**. Este, tal como o *EasyCrypt*, é uma *framework* construído sobre o *Coq proof assistant*. É baseado numa incorporação de uma linguagem de programação imperativa probabilística e numa formalização de programas probabilísticos de complexidade polinomial. Os métodos de verificação são implementados em *Coq*.

4

ALGORITMO BLAKE2

Blake2 é uma função de *hash* criptográfica baseada no algoritmo *Blake*, finalista do concurso da *NIST* para o *SHA-3*, e otimizada para ter uma melhor *performance* em *software*. Esta mostra-se mais eficiente que outros algoritmos de *hash* como *MD5*, *SHA-1*, *SHA-2* e *SHA-3* em arquiteturas *64-bits*, *x86-64* e *ARM* (16).

Este também oferece um maior nível de segurança quando comparado ao *SHA-2* e um nível similar ao *SHA-3*: imunidade a ataques de extensão do comprimento e indiferença a partir de um oráculo aleatório, entre outros (17).

A implementação do algoritmo em *Jasmin* vai ter em conta os documentos oficiais com a sua especificação e a sua implementação oficial na linguagem C (17)(18)(19).

Existem duas versões principais: *Blake2b* e *Blake2s*. A primeira está otimizada para plataformas 64-*bits*, produzindo compressões entre 1 e 64 *bytes* ; enquanto que a segunda está otimizada para plataformas desde 8-*bits* até 32-*bits*, comprimindo uma mensagem em tamanhos compreendidos entre 1 e 32 *bytes*. Visto que a *framework Jasmin* está limitada a plataformas de 64-*bits* vamos implementar apenas a versão *Blake2b* que será referida no resto deste documento apenas como *Blake2*.

4.1 CONSTANTES, PARÂMETROS E VARIÁVEIS

4.1.1 *Parâmetros*

O algoritmo tem parâmetros definidos para a sua correta execução:

Símbolo	Valor	Significado
w	64	Número de *bits* numa palavra
r	12	Número de rondas de F
bb	128	Tamanho de um bloco (em *bytes*)
outlen	$\in [1, 64]$	Tamanho de um *hash* (em *bytes*)
keylen	$\in [0, 64]$	Tamanho de uma chave (em *bytes*)
inlen	$\in [0, 2^{128}]$	Tamanho de um *input* (em *bytes*)
R_1, R_2, R_3, R_4	32 , 24 , 16 , 63	Constantes das rotações de G

Table 1: Tabela com os parâmetros do algoritmo *Blake2b*

4.1.2 *Constantes*

De forma a poder executar o algoritmo corretamente também temos de ter em conta as suas constantes:

- $IV[0..7]$ - Vetor de inicialização.
- $SIGMA[0..11]$ - Matriz de permutações de palavras. Cada posição desta matriz é uma lista 16 inteiros.

Variáveis

Além das constantes e dos parâmetros existem três variáveis que são atualizadas em cada iteração e operação do algoritmo. Todas esta alterações vão ser explicadas na próxima secção.

- $h[0..7]$ - Vetor que representa o estado atual do algoritmo. É atualizado em cada iteração da função de compressão.
- $t[0,1]$ - Vetor que representa o contador de *bytes* lidos e já processados pelo algoritmo.
- $buffer[0..15]$ - Vetor de dezasseis palavras (elementos de *64 bits*). Este vetor é preenchido de 128 em 128 *bytes* do *input* recebido e depois é processado pelo algoritmo de compressão. Este $buffer$ é de 16 elementos visto que cada elemento tem 8 *bytes* (ou *64 bits*).
- $v[0..15]$ - Vetor auxiliar para a função de mistura do algoritmo. É preenchido na função de compressão e contém o estado atual(**h**) na sua primeira metade e o vetor inicial(**IV**) na segunda.

4.2 ALGORITMOS IMPORTANTES

A função de *hash* ***Blake2*** tem dois algoritmos importantes de forma a comprimir e reproduzir um resultado que nada revela sobre os *inputs* recebidos.

Um desses algoritmos é o algoritmo de compressão(F) que irá usar o algoritmo de mistura (G) de forma a comprimir um vetor de dezasseis elementos (vetor v) num vetor de oito casas tendo em conta o estado atual do algoritmo. Os dois algoritmos (F e G) serão explicados nesta secção.

Algoritmo de Mistura

Este algoritmo caracteriza-se por receber um vetor de dezasseis palavras (elementos de *64 bits*), quatro inteiros que representam índices e duas palavras que servirão para acrescentar entropia ao algoritmo, sendo a sua assinatura: $G(v[0..15], a, b, c, d, x, y)$.

O algoritmo tem duas iterações idênticas e caracterizam-se por sucessivas adições e rotações entre as posições do vetor recebidas como argumento.

```
Function G ( v[0..16], a, b, c, d, x, y )
    v[a] += v[b] + x
```

```
    v[d] =  v[d] ^ v[a] >>> 32
    v[c] += v[d]
    v[b] = v[b] ^ v[c] >>> 24

    v[a] += v[b] + y
    v[d] = v[d] ^ v[a] >>> 16
    v[c] += v[d]
    v[b] = v[b] ^ v[c] >>> 63
```

No exemplo em cima a instrução $a >>> b$ caracteriza-se por uma rotação do valor a de b bits à direita.

Algoritmo de Compressão

O algoritmo de compressão é o principal algoritmo de todo o processo da função de *hash*. Este algoritmo é responsável por atualizar o vetor estado (h) segundo os blocos de *128 bits* vindos do *input*.

Esta função recebe um vetor de dezasseis palavras($buffer$), o vetor que representa o estado (h), o vetor que acumula o número de bytes processados (t) e um inteiro que representa um *booleano* que indica se aquele é o último bloco do *input* a ser processado.

Através do vetor estado, o vetor acumulador e a *flag* vai criar o array auxiliar v:

```
Inicializar v ( h[0..7], t[0,1], last)
    v[0..7] = h[0..7]
    v[8..15] = IV[0..7]

    v[12] ^= t[0]
    v[13] ^= t[1]

    if (last)
        v[14] ^= 0xFFFFFFFFFFFFFFFF
```

Depois vai executar 12 iterações sobre esse array, onde em cada iteração vai executar 8 vezes a função de mistura (G) de forma a transformar o vetor auxiliar. As duas palavras para causar entropia na função G são selecionadas através dos índices (obtidos através da matriz $SIGMA$) do $buffer$:

```
FOR i = 0 TO 11 DO
    s[0..15] = SIGMA[i mod 10][0..15]
    G(v, 0, 4,  8, 12, buffer[s[0]],  buffer[s[1]])
    G(v, 1, 5,  9, 13, buffer[s[2]],  buffer[s[3]])
    G(v, 2, 6, 10, 14, buffer[s[4]],  buffer[s[5]])
```

```
    G(v, 3, 7, 11, 15, buffer[s[6]],  buffer[s[7]])
    G(v, 0, 5, 10, 15, buffer[s[8]],  buffer[s[9]])
    G(v, 1, 6, 11, 12, buffer[s[10]], buffer[s[11]])
    G(v, 2, 7,  8, 13, buffer[s[12]], buffer[s[13]])
    G(v, 3, 4,  9, 14, buffer[s[14]], buffer[s[15]])
END FOR
```

No fim vamos comprimir esse *array* e atualizar o vetor que representa o estado. Como este vetor tem metade do tamanho do vetor v vamos executar a operação de *Exclusive-OR*, também conhecida como *XOR*, entre duas casas do vetor v de cada vez e depois fazer *XOR* com o estado atual de forma a atualizar o seu valor:

```
FOR i = 0 TO 7 DO
    h[i] ^= v[i] ^ v[i+8];
END FOR
```

No excerto em cima vemos que a operação *XOR*, à semelhança de muitas outras linguagens de programação, é representada pelo acento circunflexo.

Esta função F vai ser executada entre $inlen/128$ vezes, no caso do tamanho do *input* ser múltiplo de 128 e não haver chave para comprimir, ou $inlen/128+2$ vezes, no caso de não ser múltiplo de 128 e ser necessária uma compressão adicional para comprimir os *bytes* restantes, além da compressão da chave.

Algoritmo de Compressão Final

Este algoritmo é responsável por transformar o *array* estado h no *array* final com o tamanho recebido nos parâmetros da função. Este algoritmo baseia-se num ciclo que irá ser executado tantas vezes quanto o valor especificado no argumento:

```
FOR i = 0 TO outlen DO
    out[i] = h[ i >> 3 ] >> ( ( i & 7 ) * 8) & 0xFF;
END FOR
```

Esta função irá escrever os caracteres processados, já no estado final, no vetor de saída.

4.3 PROCESSO GERAL

Após termos conhecimento dos parâmetros, constantes, variáveis e funções mais importantes do algoritmo vamos agora fazer uma análise ao comportamento desta função de *hash*.

A função começa por receber seis argumentos:

- **out**: *array* onde vão ser escritos os caracteres finais, resultado de todo o processamento, a serem devolvidos.

- **outlen**: inteiro que representa o tamanho da *string* final (tamanho do *array out*). Este valor tem que ser maior que 0 e menor que 64.
- **in**: *array* de elementos recebidos como *input*. Corresponde à mensagem que queremos transformar.
- **inlen**: tamanho do *input* recebido (tamanho do *array in*).
- **key**: chave para a execução do algoritmo. Este parâmetro é opcional e a sua existência apenas tem influência no resultado final, não pondo em causa a execução (ou não) do algoritmo.
- **keylen**: tamanho da chave recebida como *input* (tamanho do *array key*). Este inteiro não poderá ser superior a 64.

O primeiro passo a fazer é inicializar as variáveis que vão ser necessárias ao longo de todo o algoritmo. Assim começamos por criar o *array* que irá representar o estado (também conhecido como h). Este vetor terá como valores iniciais os mesmos que o vetor de inicialização (IV). Após essa operação aplicamos um XOR ao primeiro elemento desse vetor, tal como explicitado no documento oficial do algoritmo.

O segundo passo é a verificação da existência de chave. Se o parâmetro que representa o tamanho da chave ($keylen$) recebido igual a 0 vamos inicializar o vetor acumulador(t) com as duas posições a 0, visto que nenhum byte foi processado. Se existir chave vamos fazer uma compressão inicial com ela atualizando assim o *array* h. Como existe um processamento de *bytes*, temos de incrementar a primeira posição do acumulador. Segundo a especificação do algoritmo este incremento terá como valor 128 (que representa um bloco inteiro) independentemente do tamanho da chave.

A partir deste momento já temos a inicialização completa. Após isso vamos percorrer o *input* em blocos de 128 *bytes* e vamos processar esses blocos através da função de compressão. Sempre que um bloco está prestes a ser processado vamos incrementar o número de *bytes* processados, que será sempre 128 com exceção do último bloco que poderão ser menos *bytes*.

Após estas consecutivas compressões ao vetor h é aplicada a função $final$ que vai transformar esse vetor no vetor final de *output*.

4.4 VERSÃO DE REFERÊNCIA

A versão de referência deste algoritmo baseou-se numa "tradução" da implementação da linguagem *C* para a linguagem *Jasmin*.

Neste processo fomos confrontados com alguns problemas devido à flexibilidade que a linguagem *C* permite mas que *Jasmin* não suporta. Assim houve um acompanhamento do documento de especificação do algoritmo (18) durante toda a construção de forma a percebermos o objetivo de uma operação e adaptar à nossa linguagem.

Outro tipo de problemas encontrados foi na alocação de registos, devido a algumas operações que necessitam de um registo específico para poderem ser executadas. As funções exportadas do programa em *Jasmin*

seguem uma interface binária de aplicação (**ABI**) e por isso assumem registos específicos. Por outro lado operações como a conjunção binária (AND ou &) exigem que a variável em questão esteja alocada no registo $\%rcx$. Ora como as funções exportadas seguem a **ABI** pode acontecer que o registo necessário para a conjunção binária já esteja a ser ocupada por um argumento da função e isso cause um erro na compilação do programa. No exemplo em baixo vemos um exemplo claro desse erro encontrado na implementação da função $final$ do algoritmo:

```
export fn blake2b(reg u64 in inlen,
                  reg u64 out outlen,
                  reg u64 key keylen)
{
    stack u64[8] h;
        ...

    // in     : %rsi
    // inlen  : %rdi
    // out    : %rdx
    // outlen : %rcx
    // key    : %r8
    // keylen : %r9

    final(out, outlen, h);
}


inline fn final(reg u64 out outlen, stack u64[8] h)
{
    reg u64 y;
        ...

    y &= 7;
        ...
}
```

No exemplo em cima vemos que o argumento $outlen$ recebido pela função exportada está alocada no registo $\%rcx$ e esse argumento também é passado à função $final$ como argumento. Nesta função a variável y vai executar a operação AND e por isso teria também de estar nesse mesmo registo. Como foi alocada noutro vai haver erro durante a compilação.

A solução para este problema passa por copiar o valor dessa variável para outra variável deixando esse registo livre para ser usado:

```
export fn blake2b(reg u64 in inlen,
                  reg u64 out outlen,
                  reg u64 key keylen)
{
    stack u64[8] h;
    reg u64 outlenCpy;
        ...

    // in     : %rsi
    // inlen  : %rdi
    // out    : %rdx
    // outlen : %rcx
    // key    : %r8
    // keylen : %r9
    outlenCpy = outlen;
    final(out, outlenCpy, h);
}
```

Após esta alteração a conjunção binária é executada e o programa é compilado com sucesso.

Depois de bastantes testes vimos que o algoritmo está corretamente implementado, isto é, com os mesmos *inputs* o resultado é o mesmo que na versão oficial na linguagem *C*. Como *inputs* este programa vai receber a mensagem *"Certificação da Componente Criptográfica: Blake2b Hash Function"*, a chave *"PassCert"* e *outlen* de 32 *bytes*:

```
C Implementation - Blake2b Hash Function("Certificação da Componente Criptográfica:
Blake2b Hash Function") with [ "PassCert" ] as key:
0b64ac38df72d1d49f53c8160346130c163db09393be6ec5ad214e11dbd731fb
```

Figure 5: Resultado da implementação da função de *hash* na linguagem *C*

```
Jasmin Implementation - Blake2b Hash Function("Certificação da Componente Criptográf
ica: Blake2b Hash Function") with [ "PassCert" ] as key:
0b64ac38df72d1d49f53c8160346130c163db09393be6ec5ad214e11dbd731fb
```

Figure 6: Resultado da implementação de referência da função de *hash* na linguagem *Jasmin*

4.5 VERSÕES ESCALAR

Depois de implementada a versão de referência passamos para a análise do código produzido para encontrar formas de o otimizar.

O primeiro passo passou pela limpeza do código. Retiramos variáveis desnecessárias e removemos funções cujas operações eram simples e não justificavam a criação de uma função apenas para essa operação.

Depois dessa limpeza optamos por tentar duplicar as instruções da função G de modo a executar logo duas misturas, reduzindo assim para metade o número das suas chamadas. Além de reduzirmos o número de chamadas da função isto permitirá tirar partido dos processadores mais modernos. Estes processadores conseguem executar operações matemáticas em paralelo quando não existe dependência entre as variáveis envolvidas usando diferentes *ALUs* (Unidade Lógica e Aritmética). Assim o algoritmo de compressão, que a utiliza, em vez de lhe passar apenas o vetor auxiliar, as quatro posições e as duas palavras vindas do *buffer*, vai passar o vetor auxiliar, oito posições e quatro palavras do *buffer* de uma só vez.

```
Function DoubleG ( v[0..16], a1, b1, c1, d1, x1, y1,
                              a2, b2, c2, d2, x2, y2 )
    v[a1] += v[b1] + x1
    v[a2] += v[b2] + x2

    v[d1] =  v[d1] ^ v[a1] >>> 32
    v[d2] =  v[d2] ^ v[a2] >>> 32

    v[c1] += v[d1]
    v[c2] += v[d2]

    v[b1] = v[b1] ^ v[c1] >>> 24
    v[b2] = v[b2] ^ v[c2] >>> 24

    v[a1] += v[b1] + y1
    v[a2] += v[b2] + y2

    v[d1] = v[d1] ^ v[a1] >>> 16
    v[d2] = v[d2] ^ v[a2] >>> 16

    v[c1] += v[d1]
    v[c2] += v[d2]

    v[b1] = v[b1] ^ v[c1] >>> 63
```

```
v[b2] = v[b2] ^ v[c2] >>> 63
```

Desta forma reduzimos o tempo de execução do algoritmo de compressão devido à redução do *overhead* total, adjacente às chamadas da função G, e à paralelização das operações matemáticas feitas pelo processador.

Versão scalar-fullstack

Desta limpeza surgiu a versão *fullstack*. Esta já deverá ser mais eficiente do que a versão de referência devido às alterações descritas em cima, mas todas as suas variáveis internas estão em *stack*, como é o caso do vetor estado (h), do vetor que representa o contador (t) e o *buffer* ($buffer$). Os valores destas variáveis apenas são carregados para registos quando são usados em alguma operação matemática, devido às características do compilador.

Versão scalar-fullmanagment

A versão escalar *fullstack* é uma versão que, à partida, já deveria ser mais eficiente que a versão de referência, dado que as alterações feitas tiveram em consideração vantagens do processador, através da paralelização, e desvantagens associadas às invocações de funções. Porém todas as variáveis do processo interno do *Blake2* estão alocadas na *stack*, mas como sabemos que acessos a registos são menos custosos do que acessos à *stack* é necessário reestruturar o algoritmo de forma a utilizar e alocar o maior número de registos possíveis. Sabendo que o número de registo das arquiteturas *x86* são 16 não podemos apenas colocar tudo em registo, sendo por isso preciso selecionar as variáveis que devemos manter em *stack* e as que podemos passar para registo.

Sendo que o *array* mais utilizado e mais acedido é o vetor representante do estado (h) essa é a principal variável a ser passada para registo. Devido ao tamanho dos vetores $buffer$ e v estes têm de se manter na *stack*. Apesar disso nem sempre é possível manter o h em registo, sendo necessário *spills* (passar de registo para *stack*) e *loads* (tirar da *stack* e guardar em registo) depois da operação.

4.6 VERSÃO FULLSTATEFULL

Muitas outras linguagens de programação já têm bibliotecas que suportam funções de *hash* criptográficas. Exemplos destas linguagens passam pelas linguagens *Python*, com a biblioteca *cryptography* (20), e *JavaScript* com as bibliotecas *Web Crypto* (21) e *Node.Js Crypto* (22).

Nestas bibliotecas as funções de *hash* seguem uma interface que se baseia na separação das diferentes fases do algoritmo escolhido. Este sistema permite aos utilizadores fazerem um número de *updates* arbitrários e só no fim fazer compressão final. Estas atualizações baseiam-se na atualização das variáveis internas consoante os *inputs* dos utilizadores. Se um utilizador quiser calcular o *hash* de um conjunto de dados, em vez de dar todos os dados de uma só vez poderá separar esse conjunto em *updates* diferentes e só no fim pedir que

o algoritmo finalize e aplique a compressão final. Esta tipo de abordagem dá uma maior liberdade e um maior poder de adaptação ao utilizador.

Na linguagem *Jasmin* não existe nenhuma função de *hash* que suporte esta tipo de abordagem por isso decidimos adaptar o algoritmo para que seja possível ao utilizador dar *updates* consecutivos e só depois finalizar e comprimir todos os dados passados ao algoritmo. Desta forma além de contribuirmos para o gestor de *passwords* contribuímos também para a comunidade do *Jasmin* quer pelo algoritmo implementado quer pelo sistema de interface de uma função de *hash* criptográfica.

Para suportamos este tipo de interação entre o algoritmo e os utilizadores temos de garantir que o estado interno do algoritmo é sempre guardado e acessível às diferentes funções em *Jasmin*. Por este motivo, o estado interno não pode estar em *stack* pois fica associado a uma função, sendo por isso necessário armazenar o estado numa zona de memória, sendo passada às diferentes funções um apontador em registo para ela.

Com as novas funcionalidades que o *Jasmin* suporta e com a ajuda do pré-processador da linguagem *C* conseguimos criar uma variável global que contém todas as variáveis de contexto do programa em *Jasmin* e através dela conseguimos suportar essa interface.

Esta variável global ($contexto$) terá de guardar um conjunto de informações necessárias para o correto funcionamento do algoritmo, e a interface que desejamos criar irá exigir algumas alterações ao código e algumas informações adicionais que antes não precisávamos, como é o caso do contador de *bytes* que o nosso *buffer* tem atualmente, visto que um utilizador pode pedir sucessivos *updates* do estado sem nunca passar mensagens que ocupem todo o espaço do *buffer*. Isto também exigirá algumas alterações aos procedimentos em si.

Antes invocávamos a função de compressão apenas na inicialização, no caso de haver chave, e no função $update$, que atualizava o estado, depois de copiarmos 128 *bytes* de cada vez para o nosso *buffer*. Agora alteramos esse procedimento de forma a conseguir comprimir toda a mensagem. Como o utilizador pode invocar a função de *update* e não fornecer uma mensagem que ocupe todo o *buffer*, é necessário que quando ele invoque a função de finalização haja uma verificação do contador de *bytes* do *buffer*, para nos certificarmos que nenhuma parte da mensagem fica por atualizar e comprimir.

Outra alteração ocorreu na própria função de atualização. Agora sempre que esta função é invocada, até que a mensagem a processar acabe, ele recorre a uma função auxiliar que vai preencher o *buffer* e invocar a função de compressão até não haver mais mensagem. A diferença está que agora, sempre que ele recebe uma mensagem, consulta o contador do *buffer* para saber o tamanho de *bytes* da mensagem que pode utilizar para o preencher e apenas preenche o *buffer*. A compressão só ocorre quando o contador está a 128, antes da cópia da mensagem para o *buffer*.

Para isto ser possível a variável contexto é um vetor de 28 elementos de 8 *bytes*. Neste vetor temos várias variáveis necessárias para o correto funcionamento do algoritmo, sendo que a cada posição está associado uma variável em específico, e iremos descrever a sua composição.

Os primeiros 16 elementos vão corresponder ao *buffer* que vai sendo atualizado com a/as mensagem/mensagens.

```
contexto[0..15] : buffer
```

Os 8 elementos seguintes correspondem ao estado interno do programa. À medida que são executadas compressões o estado é atualizado e guardado dentro deste espaço do contexto.

```
contexto[16..23] : estado interno do programa
```

Os 2 elementos seguintes correspondem ao número de *bytes* já processados. O primeiro elemento vai acumulando o número total e o segundo elemento apenas é incrementado se existir um caso de *carry overflow* devido ao valor máximo de um inteiro de 8 *bytes*.

```
contexto[24,25] : contadores dos \textit{bytes} já processados
```

Depois temos o contador do *buffer*. Devido às alterações aplicadas para ser possível criar uma interface para a função de *hash*, e sabendo que um *buffer* apenas é comprimido quando estiver completo, é necessária uma variável que contém o espaço já ocupado. Sempre que existe uma compressão, este contador passa a ser 0.

```
contexto[26] : contador atual do buffer
```

Na última posição temos o tamanho do *output*. Como é necessário o tamanho de *output* em alguns métodos aproveitamos para o inserir no contexto, permitindo assim acrescentar algumas funcionalidades para garantir o bom funcionamento do algoritmo. Através deste elemento vai existir uma confirmação do estado do programa, sendo que se o *output* for igual a 0 significa que o contexto ainda não foi inicializado ou já foi finalizado, não sendo por isso permitido executar novos *updates*.

```
contexto[27]: tamanho do output
```

Desta forma a nossa interface fica com as seguintes assinaturas:

```
blake2b_init(ctxt, outlen, key, keylen);

blake2b_update(ctxt, input, inputlen);

blake2b_finish(ctxt, output);
```

Como vemos pelas nomenclaturas em cima a inicialização é executada e apenas passamos como argumento o apontador para o contexto, o tamanho do vetor de saída, o apontador para a chave e o tamanho da chave. No fim dessa execução o nosso contexto terá sido atualizado. Depois podemos executar múltiplas atualizações invocando a função de *update* que irá atualizar o contexto, passando-o como argumento assim como a mensagem e o seu tamanho. Por fim a função de finalização será invocada e passamos como argumento apenas o contexto e o apontador para o vetor de *output*.

4.7 VERSÃO OTIMIZADA

Depois de uma versão que suporta uma interface típica de uma função de *hash* decidimos tirar partido de uma implementação eficiente na linguagem *C* para otimizarmos a nossa versão também.

No repositório oficial do *Blake2* existe uma versão otimizada que suporta *SSE* e *AVX*. Esta versão caracteriza-se por utilizar instruções *Intrinsic* da *Intel*, aplicadas à função de compressão e suas auxiliares.

Inspirando-nos nesta versão utilizamos o *Intel Intrinsics Guide* (23), que é uma ferramenta que serve como guia para as instruções *Intrinsics*. Este guia explica o que cada instrução faz e ainda nos fornece a primitiva *Assembly* que a representa. Através destas primitivas alteramos o nosso programa de forma a otimizar a sua execução mantendo as suas funcionalidades.

De forma a executar estas otimizações na função de compressão, em vez de trabalharmos com inteiros de 64 *bits*, são usados inteiros de 128 *bits*, o que implica uma abordagem diferente ao algoritmo.

Como vimos no capítulo 4.2 a função de compressão trabalha com inteiros de 64 *bits* e divide-se em três tarefas:

- Inicializar vetor auxiliar v
- 12 *Rounds* (invocando em cada *Round* 8 vezes o algoritmo de *mistura* G)
- Comprimir o vetor auxiliar v, atualizando o estado do programa h

Agora como trabalhamos com inteiros de 128 *bits* vamos dispensar o vetor auxiliar v e transformar os seus valores em 8 inteiros de 128 *bits* da seguinte maneira:

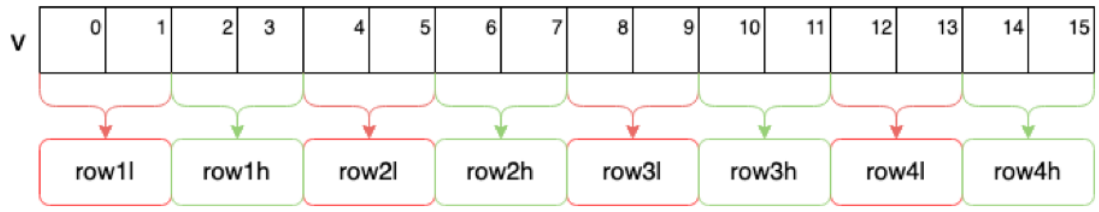

Figure 7: Transformação do array v em 8 variáveis de 128 *bits*

A imagem em cima representa apenas a correspondência entre o vetor v e as novas variáveis. Para a sua inicialização são usadas leituras diretas do estado do programa, atribuições (quando são constantes) ou instruções *Assembly*. Cada uma destas novas variáveis está no formato *little-endian*.

Depois de inicializadas estas 8 variáveis são invocadas também os 12 *Rounds*, onde cada *Round* atualiza as 8 *rows*. Cada *Round* é bastante distinto dos *Rounds* das outras versões. Enquanto que nas versões de referência ou escalares cada *Round* aplicava 8 misturas (ou 4 nas versões escalares), através da função G, nesta versão otimizada vão ser executadas 4 misturas, mas recorrendo a duas funções de mistura diferentes, a uma diagonalização e à sua inversa. A função de mistura G vai deixar de ser utilizada e vão ser usadas duas novas funções de mistura $G1$ e $G2$.

No capítulo 4.2 vemos como o algoritmo de mistura funciona. Estas novas funções $G1$ e $G2$ vão ser o resultado de partir esse algoritmo em 2, sendo que o $G1$ corresponde à primeira mistura de cada posição, e a função $G2$ à segunda. A razão pela qual estas vão ser repartidas é porque as mensagens, vindas do *buffer*, vão ser obtidas e passadas de forma diferente, sendo necessário partir estas duas operações.

Quanto à função de diagonalização e a sua respetiva inversa, inspiramo-nos na versão oficial otimizada na linguagem *C* e com a ajuda do *Intel Intrinsics Guide*, obtivemos as instruções *Assembly* correspondentes e adaptamos ao *Jasmin*.

Estas quatro funções vão ser utilizadas em cada *Round* seguindo a seguinte sequência de instruções:

```
b0 = ...; b1 = ... ;
G1(row1l,row2l,row3l,row4l,row1h,row2h,row3h,row4h,b0,b1);

b0 = ...; b1 = ... ;
G2(row1l,row2l,row3l,row4l,row1h,row2h,row3h,row4h,b0,b1);

DIAGONALIZE(row1l,row2l,row3l,row4l,row1h,row2h,row3h,row4h);

b0 = ... ; b1 = ... ;
G1(row1l,row2l,row3l,row4l,row1h,row2h,row3h,row4h,b0,b1);

b0 = ... ;b1 = ... ;
G2(row1l,row2l,row3l,row4l,row1h,row2h,row3h,row4h,b0,b1);

UNDIAGONALIZE(row1l,row2l,row3l,row4l,row1h,row2h,row3h,row4h);
```

De realçar que as variáveis $b0$ e $b1$ estão com o valor ... porque cada *Round* tem a sua própria instrução de obter essas variáveis com diferentes posições do *buffer*.

Depois da execução dos 12 *Rounds* a atualização do estado interno do programa é feita tal como as outras versões anteriores, com a nuance de agora não trabalharmos com o vetor v, mas sim com as 8 variáveis row. A atualização em si é feita através de *XORs* e, como trabalhamos com variáveis de 128 *bits*, são atualizadas duas posições de cada vez:

```
h[0,1] ^= row1l ^ row3l ;
h[2,3] ^= row1h ^ row3h ;
h[4,5] ^= row2l ^ row4l ;
h[6,7] ^= row2h ^ row4h ;
```

Função de Diagonalização e sua Inversa

A função de *diagonalização* caracteriza-se por misturar as variáveis de 128 *bytes*, não só entre elas, mas também os próprios valores, ou seja, não são apenas feitas trocas diretas.

Para tirar partido das instruções vetorizadas é necessário ter o estado representado de duas maneiras diferentes. Como vimos pela figura 7, cada uma destas variáveis toma o valor de duas posições do antigo vetor v. De modo a perceber as alterações que esta função faz nas variáveis row vamos ter sempre em atenção esta comparação entre elas e as posições que ocupariam no vetor v.

Antes desta função ser aplicada as oito variáveis assumem os seguintes valores:

Figure 8: Variáveis row no início da função de diagonalização

Nesta figura 8 temos cada variável row repartida pelos valores de v que a constituem. Tendo em conta que as variáveis estão em formato *little-endian* sabemos que os 8 *bytes* menos e mais significativos estão à esquerda e à direita, respetivamente.

Agora que sabemos os valores iniciais das variáveis temos de ter em conta as duas principais instruções *Assembly* que utilizamos: *INTERLEAVE_QWORDS* e *INTERLEAVE_HIGH_QWORDS*. A primeira recebe como argumento dois inteiros de 16 *bytes*, pega nos 8 *bytes* menos significativos do primeiro argumento e usa-os como os *bytes* menos significativos do *output*, e depois pega nos 8 *bytes* menos significativos do segundo argumento e usa-os como os *bytes* mais significativos do *output*:

```
INTERLEAVE_QWORDS(src1[127:0], src2[127:0]) {
        dst[63:0] := src1[63:0]
        dst[127:64] := src2[63:0]
        RETURN dst[127:0]
}
```

Já a segunda função recebe o mesmo número e tipo de argumentos mas em vez de utilizar os *bytes* menos significativos, utiliza os mais significativos:

```
INTERLEAVE_HIGH_QWORDS(src1[127:0], src2[127:0]) {
    dst[63:0] := src1[127:64]
        dst[127:64] := src2[127:64]
        RETURN dst[127:0]
}
```

Posto isto poderemos agora ver como esta função de diagonalização se comporta.

Ela começa por executar uma troca direta entre as variáveis $row3h$ e $row3l$.

Depois vai atuar sobre as variáveis $row4l$ e $row4h$. Primeiro cada uma delas começa por inverter os seus *bytes* (os mais significativos passam a ser os menos significativos e vice-versa). Seguidamente as duas variáveis trocam os seus *bytes* menos significativos entre eles.

O mesmo acontece entre as variáveis $row2h$ e $row2l$.

No fim desta função teremos as variáveis com os seguintes valores, quando comparados ao vetor v:

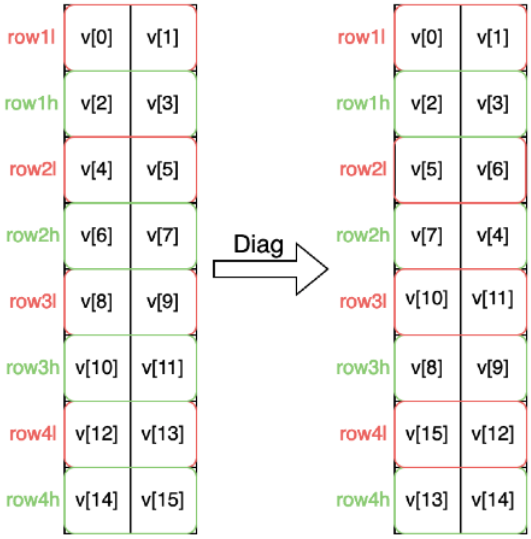

Figure 9: Variáveis row no fim da função de diagonalização

Já a função inversa trata de usar as mesmas funções a cima referidas, mas para inverter para as posições iniciais.

Tal como no processo de diagonalização, começamos por trocar as variáveis $row3l$ e $row3h$.

Depois invertemos a significância dos *bytes* nas variáveis $row2l$ e $row2h$ e trocamos os *bytes* mais significativos entre eles.

Por último fazemos a mesma coisa com as variáveis $row4h$ e $row4l$, obtendo o resultado:

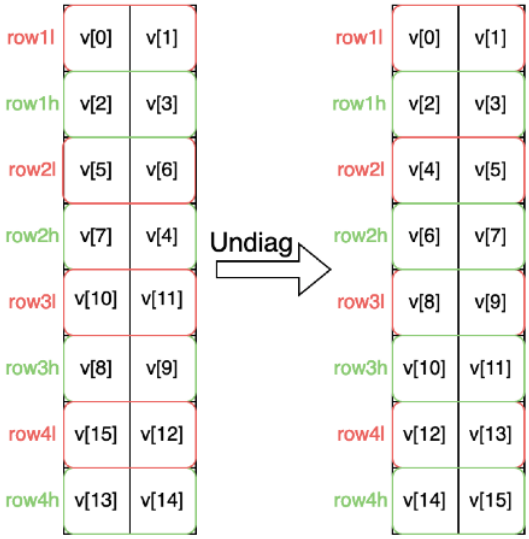

Figure 10: Variáveis *row* no fim da função inversa à diagonalização

Através destas funções, juntamente com as funções de mistura $G1$ e $G2$ conseguimos replicar a função de compressão normal utilizando instruções *Assembly* para tornar o nosso algoritmo ainda mais eficiente.

5

ALGORITMO ARGON2

O algoritmo *Argon2* pertence às primitivas criptográficas de *Key Derivation Functions* e é um dos algoritmos disponíveis no *KeePass*, juntamente com o algoritmo *AES-KDF*.

Este algoritmo caracteriza-se por produzir uma *tag* como *output* depois de receber uma mensagem, um *nonce* e alguns parâmetros que mais à frente vamos demonstrar. Esta *tag* servirá depois como chave para operações de encriptação dos dados presentes na base de dados do gestor de *passwords*.

Como já foi referido o *Argon2* usa a função de *hash Blake2b* para operações internas.

Vamos começar por fazer uma explicação muito breve de como este algoritmo funciona.

5.1 PROCESSO GERAL

De acordo com o seu documento de especificação (24) o *Argon2* comporta-se de forma muita simples: este executa a sua função de compressão G, quantas vezes foram necessárias até esgotarem a mensagem de *input*, onde recebe blocos de mensagem de 1024 *bytes*, aplica a sua função de compressão à mensagem obtendo um *output* de 1024 *bytes* que irá atualizar o seu estado interno.

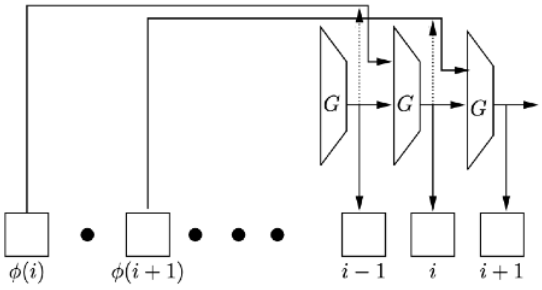

Figure 11: Processo geral interno da *KDF Argon2*

No fim das funções de compressão é usada a função de *hash*, que neste caso é a *Blake2b*, para produzir a *tag* com o tamanho desejado.

5.2 IMPLEMENTAÇÃO EM JASMIN

Depois de analisado o processo geral do *Argon2*, segundo a sua especificação (25) vimos que este não se adequa a ser implementado em *Jasmin*. Este algoritmo exige a manipulação dinâmica da memória, como alocação e libertação, que não é possível realizar em *Jasmin*, pois não suporta bibliotecas do sistema operativo que são utilizadas na implementação.

A estratégia passa agora por detetar rotinas que poderão ser adaptadas para *Jasmin*, assim como integrar a nossa versão da função de *hash Blake2b* nesta *Key Derivation Function*.

Depois de uma análise ao código em si (26), pudemos perceber duas coisas: que as operações adequadas a serem implementadas em *Jasmin* eram apenas operações matemáticas entre blocos (como somas, *XORs*, *etc*) e que o algoritmo tem uma versão própria da função de *hash Blake2b*, que tira proveito das diferentes operações (*init*, *update* e *final*) que a constituem, para gerar o seu próprio *output*, denominada como $Blake2b\_long$.

Sabendo isto decidimos numa primeira fase apenas substituir as diferentes operações do *Blake2b* pelas nossas, e depois implementamos todo o algoritmo.

5.3 BLAKE2B LONG

Esta versão *long* do Blake2b caracteriza-se por permitir tamanhos de *output* arbitrários, o que se diferencia da versão clássica que só permite tamanhos de saída até 64 *bytes*, e por dispensar o uso de chave para o processo.

Sendo que eles utilizam as primitivas básicas do *Blake2b* clássico, o tamanho do *output* utilizado nas funções de $init$ e $final$ está limitado pelos 64 *bytes*, o que implica que haja novos procedimentos, fora dessas primitivas, para que consiga suportar *outputs* de qualquer tamanho.

Esta nova versão tem dois comportamentos distintos consoante o tamanho do vetor de saída desejado. Se este estiver dentro do espectro permitido pelo *Blake2b* o *output* é obtido através da atualização do estado com o tamanho de *output* e depois a mensagem, tal como podemos ver no seguinte excerto:

```
if(outlen <= 64) {
    blake2b_init(state, outlen);

    outlen_bytes <- bytes(outlen);
    blake2b_update(state,outlen_bytes, sizeof(outlen_bytes));
    blake2b_update(state,in,inlen);

    blake2b_final(state,out,outlen);
}
```

Quando o tamanho de saída é maior que 64 *bytes* vão ser invocadas várias vezes a função de *hash*, onde o resultado de cada execução é usado na iteração seguinte e são guardados os 32 *bytes* mais significativos no vetor de saída, sendo que a primeira execução é igual ao excerto demonstrado em cima.

```
else {
    int toproduce;
    int out_buffer[64]; int in_buffer[64];

    blake2b_init(state, 64);
    outlen_bytes <- bytes(outlen);
    blake2b_update(state,outlen_bytes, sizeof(outlen_bytes));
    blake2b_update(state, in,inlen);
    blake2b_final(state, out_buffer, 64);

    memcpy(out,out_buffer,32); //memcpy(dest,source,size)
    out <- out + 32;
    toproduce <- outlen - 32;

    while(toproduce > 64) {
        in_buffer <- out_buffer ;

        blake2b_init(state, 64);
        blake2b_update(state, in_buffer, 64);
        blake2b_final(state, out_buffer, 64);

        memcpy(out,out_buffer,32);
        out <- out + 32;

        toproduce <- toproduce - 32;
    }

    if(toproduce > 0) {
        in_buffer <- out_buffer ;

        blake2b_init(state, toproduce);
        blake2b_update(state, in_buffer, 64);
        blake2b_final(state, out_buffer, toproduce);

        memcpy(out,out_buffer,toproduce);
```

```
    }
}
```

5.3.1 *Integração*

Como já foi referido antes, numa primeira fase apenas adaptamos a implementação em *C* para suportar a nossa interface em *Jasmin*, o que se tornou um processo bastante simples visto que não tínhamos de tratar da gestão e cópias de memória. A implementação deste algoritmo ficou bastante idêntica à original e foram apenas substituídas as chamadas aos diferentes métodos do *blake2b* pela nossas implementações.

Depois de feita esta integração decidimos tentar implementar todo o algoritmo na linguagem *Jasmin*.

5.3.2 *Implementação em Jasmin*

Nesta fase já tínhamos todos os processos relacionados com a função de *hash* implementados e completamente funcionais, sendo que o desafio aqui seria tratar das cópias de memória e gestão dos registos para que estes não se esgotem durante a execução do algoritmo. Vamos agora fazer uma análise detalhada ao código resultante.

Antes de percorrermos o código vamos olhar para a assinatura da função:

```
fn blake2b_long_jazz(reg u64 ctxt0 out0 outlen0 input0 inputlen0)
-> reg u64
```

A função precisa de receber os apontadores para o contexto, vetor de saída e mensagem, e os tamanhos de saída e da mensagem.

O algoritmo começa por criar os registos necessários para guardar os valores passados por argumento e depois variáveis auxiliares que vão ser necessárias. Devido à limitação do número de registos estas variáveis poderão ser usadas para qualquer informação:

```
    reg u64 ctxt out outlen input inputlen;
    reg u64 r t1 t2 t3;
```

Depois disso vamos criar variáveis *stack* para serem usadas em operações *spill* para ganharmos espaço para operações em registo:

```
    stack u64 out_s input_s inputlen_s;
```

Declaradas todas as variáveis vamos agora resolver algumas restrições da *ABI* copiando os valores dos argumentos para as variáveis criadas na função:

```
ctxt=ctxt0; outlen=outlen0;
```

Nesta fase inicial vamos também fazer *spill* dos argumentos que não vão ser necessários:

```
out_s = out0;
input_s = input0;
inputlen_s = inputlen0;
```

A partir daqui vamos começar a preparar as variáveis para invocarmos a inicialização do estado. Como temos os dois comportamentos diferentes temos de executar uma verificação do tamanho de saída e guardar o valor correto numa variável auxiliar para ser usada na chamada ao $init()$. Como o método clássico do $init$ exige uma chave e seu respetivo tamanho, enquanto que esta versão não o permite, vamos usar duas variáveis auxiliares para dizer ao método de inicialização que não existe chave, atribuindo-lhes o valor 0:

```
if (outlen0 <= 64) {
    t3 = outlen0;
}
else {
    t3 = 64;
}

t1 = 0; t2=0;
```

Tendo todas as variáveis necessárias vamos agora invocar a inicialização:

```
blake2b_init(ctxt, t3, t1, t2);
```

Como vimos no capítulo 5.3 é necessário atualizar o estado com o tamanho de saída. Para tal em *Jasmin* vamos usar uma variável que vai assumir o apontador para o estado e depois vai avançar 216 *bytes*, o que corresponde à posição correspondente ao *outlen*. O tamanho desse valor é de 4 *bytes* e podemos agora atualizar o estado. De realçar que antes disso, temos de atualizar o contexto para o tamanho de saída desejado pelo utilizador, pois a operação de $init()$ altera essa variável durante a sua execução.

```
OL(ctxt) = outlen;
t1 = ctxt;
t1 += 8*27; // OL(ctxt)
t2 = 4;
blake2b_update(ctxt, t1, t2);
```

O passo seguinte trate-se de atualizar o estado interno com a mensagem recebida como *input*. Como anteriormente fizemos *spill* do apontador no qual a mensagem foi escrita, agora é necessário carregá-la para um registo através de um *load* e depois invocar a função de *update* outra vez:

```
input=input_s; inputlen=inputlen_s;
blake2b_update(ctxt, input, inputlen);
```

Estando a primeira parte tratada, vamos agora criar um ciclo que irá invocar os processos do *blake2* e guardar os 32 *bytes* mais significativos no *output*, até obter o tamanho de saída desejado.

Antes de iniciar o ciclo vamos buscar a nossa variável de contexto que contém o valor do *output* para ser usada como condição de permanecer ou sair do mesmo.

Visto que já foram executadas as primeiras inicializações e atualizações do estado, começaremos por carregar o apontador do *output* e invocar o método de finalização. Contrariamente às outras versões, as operações matemáticas executadas neste método de finalização não vai ser executadas nela, mas sim numa função auxiliar $memcpy()$. Esta função foi necessária devido à copia dos 32 *bytes* mais significativos para o *output* e do resultado total para a próxima iteração do ciclo:

```
  inline fn memcpy(reg u64 dest source size) {
reg u64 i size8 tmp8;
reg u8 tmp;

size8 = size;
size8 >>= 3;
i = 0;
while(i < size8) {
  tmp8 = [source + 8*i];
  [dest + 8*i] = tmp8;
  i += 1;
}
i <<= 3;
while (i < size) {
  tmp = (u8) [source + i];
  (u8) [dest + i] = tmp;
  i += 1;
}
```

Anteriormente a função de finalização verificava se restavam *bytes* por processar no *buffer* e só depois dessa verificação, e respetiva última atualização se houver *bytes*, é que o método cumpria o seu propósito: expandir o seu estado interno para o *output* com o tamanho desejado. Agora essa expansão é feita neste novo método.

Deste modo o método de finalização nesta versão *long* começa por verificar o tamanho de *output* guardado no seu contexto. Se este for menor que 64 *bytes* ele invoa este método passando como argumentos o destino (o vetor de saída), a fonte (apontador para o estado interno) e o tamanho de *output*. No caso de ser maior que 64 *bytes* ele assume que o tamanho a copiar serão apenas 32 *bytes*, obtendo assim os 32 *bytes* mais significativos:

```
outlen = OL(ctxt);
while (outlen > 64) {
    out=out_s;
    blake2b_finish(ctxt, out);
}
```

Agora temos de utilizar os 64 *bytes* da expansão do estado interno para um *buffer* e voltar a comprimir. Tendo em conta as alterações referidas em cima vamos apenas utilizar uma variável que vai guardar o apontador para o estado interno do programa e outra com o tamanho necessário (*64 bytes*), invocando o método $memcpy()$ que irá guardar o resultado diretamente no *buffer* do contexto:

```
outlen = OL(ctxt);
while (outlen > 64) {
    out=out_s;
    blake2b_finish(ctxt, out);

    t1 = ctxt;
    t1 += 8*16; //apontador para o buffer
    t2 = 64;
    memcpy(ctxt, t1, t2);
}
```

Estando atualizado o *buffer* vamos avançar o *output*, dar *spill* ao seu apontador e reduzir o número de *bytes* que faltam processar:

```
outlen = OL(ctxt);
while (outlen > 64) {
    out=out_s;
    blake2b_finish(ctxt, out);
    t1 = ctxt;
    t1 += 8*16; //apontador para o buffer
    t2 = 64;
    memcpy(ctxt, t1, t2);

    out += 32;
```

```
        out_s = out;
        outlen -= 32;
    }
```

Vamos agora proceder à nova iteração da função de *hash* e para isso temos de inicializar novamente o estado e o contador de *bytes* do *buffer* a 64 para indicar que ele já está preenchido:

```
    outlen = OL(ctxt);
    while (outlen > 64) {
        out=out_s;
        blake2b_finish(ctxt, out);
        t1 = ctxt;
        t1 += 8*16; //apontador para o buffer
        t2 = 64;
        memcpy(ctxt, t1, t2);
        out += 32;
        out_s = out;
        outlen -= 32;

        t1 = 0; t2=0; t3=64;
        if ( outlen <= 64) { t3 = outlen; }
        blake2b_init(ctxt, t3, t1, t2);

        C(ctxt) = 64;
    }
```

Não será necessário invocar o método de $update()$ visto que o método de $finish()$ verifica se ainda há bytes por processar no *buffer*. Como a última instrução de cada iteração do ciclo atualiza o contador de *bytes* do *buffer* para 64, a função de finalização manda executar a atualização do estado interno antes de executar a cópia dos *bytes*.

Saído do ciclo vamos carregar o nosso apontador para o vetor de saída e pedir uma última finalização para copiar os *bytes* que poderão restar:

```
    out=out_s;
    blake2b_finish(ctxt, out);
```

Desta forma temos o algoritmo *blake2b_long* completamente implementada em *Jasmin*, podendo ser integrada nas *APIs* em*C* com uma única chamada a este método, poupando ao programa as cópias de memória e as chamadas aos diferentes métodos da interface. É esperado que esta versão seja mais eficiente que a versão em *C* e a versão integrada descrita na secção 5.3.1.

6

VERIFICAÇÃO

6.1 SAFETY CHECK

Umas das características do *Jasmin* é que este possui ferramentas que nos ajudam no processo de verificação e de garantias de segurança. Uma dessas ferramentas é o *safety checker* que nos permite pedir ao compilador que execute uma verificação sobre o nosso programa para analisar se este é seguro ou não, e ainda apontar possíveis falhas de segurança da nossa implementação.

Este *safety checker* caracteriza-se por ser um analisador estático que vai percorrer o código em *Jasmin*, resultante do pré-processamento, e vai tentar verificar se as funções a serem exportadas são seguras. Podemos considerar se um programa é seguro ou não através das seguintes propriedades:

- **terminação** - independentemente do comportamento de uma função, é sempre garantido que ela termina e que não fica "presa" em alguma instrução (como por exemplo um ciclo sem saída).
- **acessos a vetores são executados dentro dos seus limites** - sempre que são executados acessos a vetores locais ao programa, quer para leitura quer para escrita, apenas são executadas em posições dentro do tamanho do mesmo.
- **acessos a posições de memória válidas** - além do acesso a vetores, por vezes precisamos de aceder a posições de memória que deverão ser válidas, isto é, ter sido uma posição alocada e estar devidamente alinhada (como é o caso de acesso a variáveis passadas por apontadores).
- **operações aritméticas aplicadas a argumentos válidos** - garantir que a qualquer operação aritmética usada são-lhe passados argumentos válidos.

A garantia destas propriedades resulta em funções que a partir de um estado de memória inicial, independentemente dos argumentos e do seu comportamento em concreto, levam sempre a uma execução de um conjunto de instruções que terminam sempre em um estado final válido, ou seja, uma execução segura.

Para verificarmos se o nosso programa em *Jasmin* é seguro vamos então tirar proveito desta ferramenta aplicando-a à nossa versão *fullstatefull*, que contém a interface típica de uma função de *hash*. Para tal temos de usar uma *flag* específica do compilador do *Jasmin*, que irá ser aplicada ao ficheiro de código já pré-processado pelo pré-processador do *C*:

```
jasminc Blake2b.japp -checksafety
```

Uma das características deste analisador é que nos permite informá-lo se algum dos *inputs* de uma determinada função são apontadores ou apenas tamanhos, através da opção *safetyparam* que tem o seguinte aspeto:

```
-safetyparam "funcao > apontador1, apontador2; tamanho1, tamanho2"
```

Como a nossa interface suporta três funções diferentes, vamos usar três vezes a opção de *safetyparam* para indicarmos os apontadores e os tamanhos que são passados como argumento de cada função:

```
jasminc Blake2b.japp -checksafety
    -safetyparam "blake2b_init_jazz > ctxt, key; outlen, keylen"
    -safetyparam "blake2b_update_jazz > ctxt, input; inputlen"
    -safetyparam "blake2b_finish_jazz > ctxt, output"
```

Quando aplicada esta funcionalidade a um programa em Jasmin este terá um formato composto por:

- Função que está a analizar.
- Possíveis violações de segurança, as quais o compilador não consegue dar garantias.
- Intervalos de memória válidos para variáveis.
- Pré-condição da região de memória alocada segura, por meio de uma conjunção de desigualdades lineares.
- Variáveis que devem ter memória alinhada e respetivos tamanhos.

Após executarmos as respetivas verificações e análises à nossa implementação, obtivemos os seguintes *outputs* que apenas iremos apresentar a informação mais relevante. Optamos por não demonstrar as pré-condições das regiões de memória pois estas apenas revelam que as nossas variáveis que representam apontadores têm um tamanho mínimo e máximo (tipicamente definidos pelos limites da própria máquina). Os alinhamentos também não são relevantes pois a própria especificação do algoritmo já determina essa condição. Assim sendo vamos apenas demonstrar as possíveis violações de segurança pois apenas essas demonstram informação útil para os aspetos que queremos apresentar.

Comecemos então por analisar o resultado desta funcionalidade aplicada à função de inicialização:

```
Default checker parameters.

Analyzing function blake2b_init_jazz

*** Possible Safety Violation(s):
  line 190: is_valid buf.320 + (((64u) 8) * i.323) u64
  line 190: aligned pointer buf.320 + (((64u) 8) * i.323) u64
```

```
  line 198: is_valid buf.320 + i.323 u8
  line 198: aligned pointer buf.320 + i.323 u8

Program is not safe!
```

Como podemos ver pelo excerto em cima, a ferramenta não consegue dar garantias que a função é segura pois existem possíveis violações de segurança. Esta análise não consegue dar certezas se o comportamento da nossa função é seguro e alerta-nos para possíveis acessos a zonas de memória que não são válidas para uma variável, e que não pode confirmar se a zona de memória de uma variável está devidamente alinhada.

Vamos agora analisar cada alerta recebido comparando com o nosso código para tirar uma conclusão sobre o resultado obtido.

Todas as violações ocorrem na mesma função:

```
fillup0Buff(reg u64 buf buflen)
```

,que recebe como argumentos o apontador para o *buffer* e o tamanho do *buffer* já preenchido. O objetivo desta função é preencher o resto do *buffer* com 0's, para este ficar completo e poder ser processado.

As duas primeiras violações ocorrem na instrução de colocar um 0 numa dada posição do *buffer*, e esta vai ser executada num ciclo:

```
i = 0;
while ( i < n8 )
{
    [buf + 8*i] = 0; // line 190
    i += 1;
}
```

, onde a variável $n8$ é obtida através de um *right shift* de 3 casas do tamanho que falta preencher no *buffer*, para obtermos o número de posições de 8 *bytes* ainda disponíveis. Visto que este é um vetor onde cada posição é representado por valores de 8 *bytes*, através deste ciclo vamos preencher 8 *bytes* sempre que possível.

Essa é também a razão do alerta. O compilador não consegue identificar que valores é que a nossa variável $n8$ vai assumir, e como o nosso *buffer* é um apontador, o compilador não tem certeza se estamos a aceder a posições válidas e se a zona de memória alocada para o *buffer* está corretamente alinhada.

O mesmo acontece com as outras duas possíveis violações, onde agora percorremos o *buffer byte* a *byte*, quando necessário, para preenchermos o restante espaço:

```
i <<= 3;
while ( i < t )
{
    (u8)[buf + i] = 0; //line 198
```

```
    i += 1;
}
```

, onde o nosso *i* vai agora sofrer um *left shift* para obtermos o número de *bytes* já preenchidos e a variável *t* é uma variável que guarda o tamanho máximo do *buffer*: 128 *bytes*. Através deste ciclo vamos atualizar os *bytes* restantes com o valor 0.

Tal como o caso em cima, o compilador não consegue ter a certeza se estamos a aceder a posições de memória válidas e devidamente alinhadas, e por isso alerta-nos para essa situação.

Apesar disso estes alarmes não implicam que o programa não seja de facto seguro. Este processo de *shifts* lógicos e preenchimento de *buffer* através de apontador é assegurado através da especificação *RFC* e através de uma cuidadosa análise aos valores que as nossas variáveis poderiam assumir para não acederem a posições inválidas.

Após a análise ao método de inicialização vamos agora analisar o método de *update*.

```
Default checker parameters.

Analyzing function blake2b_update_jazz

*** Possible Safety Violation(s):
  line 217: is_valid input.315 + (((64u) 8) * i.314) u64
  line 217: aligned pointer input.315 + (((64u) 8) * i.314) u64
  line 218: is_valid buf.317 + (((64u) 8) * i.314) u64
  line 218: aligned pointer buf.317 + (((64u) 8) * i.314) u64
  line 225: is_valid input.315 + i.314 u8
  line 225: aligned pointer input.315 + i.314 u8
  line 226: is_valid buf.317 + i.314 u8
  line 226: aligned pointer buf.317 + i.314 u8
  line 273: termination

Program is not safe!
```

Tal como no caso anterior é importante perceber que método e instruções estão a causar alertas e pudemos verificar que as oito primeiras ocorrem na função:

```
fillBuff(reg u64 buf buflen input inputlen)
```

Analisando a função $fillBuff$ esta é bastante semelhante à função $fillup0Buff$, mas agora, em vez de preencher o *buffer* com zeros, vamos preenche-lo com a mensagem passada por *input*:

```
i = 0;
while ( i < n8 )
```

```
{
    t = [input + 8*i]; //line 217
    [buf + 8*i] = t; //line 218
    i += 1;
}
i <<= 3;

while ( i < n )
{
    t8 = (u8)[input + i]; //line 225
    (u8)[buf + i] = t8; //line 226
    i += 1;
}
```

Tal como no caso anterior, a ferramenta não consegue ter a certeza que posições estamos a aceder e por isso lança um alerta para possíveis acessos ilegais a zonas de memória que poderão também não estar alinhadas.

Já o erro de terminação ocorre na própria função de $update$ a ser exportada:

```
blake2b_update_jazz(reg u64 ctxt input inputlen)
```

Este alerta de terminação informa que poderá haver a possibilidade de esta função nunca terminar, caso não alcance uma determinada cláusula. Se olharmos para o código em si vemos que o erro está numa cláusula $else$ que executa um ciclo $while$:

```
else{ //line 273
    while (inputlen > 0) {
        input, inputlen = updateBuff(ctxt, input, inputlen);
    }
    ret = 0;
 }
```

Este ciclo $while$ é executado enquanto que o tamanho da mensagem recebida for maior que zero, e em cada execução apenas invoca a função $updateBuff$, que se caracteriza por atualizar o *buffer*. Esta função retorna o apontador para a mensagem atualizado e o tamanho da mensagem que ainda falta processar. Sabendo isto, percebemos que a nossa variável $inputlen$ decresce a cada iteração do ciclo e que eventualmente irá terminar em zero, o que significa o término do próprio ciclo e consequentemente o término desta função.

O alerta recebido será porque o compilador não consegue identificar os valores que o $inputlen$ poderá assumir e como não existe nenhuma variável que explicitamente decresce no ciclo $while$.

Analisando agora a função de finalização obtivemos o seguinte *output*:

```
Default checker parameters.

Analyzing function blake2b_finish_jazz

*** Possible Safety Violation(s):
  line 190: is_valid buf.320 + (((64u) 8) * i.323) u64
  line 190: aligned pointer buf.320 + (((64u) 8) * i.323) u64
  line 198: is_valid buf.320 + i.323 u8
  line 198: aligned pointer buf.320 + i.323 u8

Program is not safe!
```

Esta parte do *output* é igual ao obtido na função de inicialização e por isso facilmente percebemos que sempre que invocarmos a função de $fillup0Buff$ vamos obter alertas de possíveis acessos ilegais.

6.2 CONSTANT-TIME

Outra das propriedades mais relevantes do *Jasmin* é que este permite extrair para *EasyCrypt* um modelo base para provas de *constant-time*.

A propriedade de *constant-time* sobre um programa diz-nos que nem o fluxo de instruções (desde blocos condicionais a ciclos) ou acessos de memória depende de dados sensíveis (segredos).

Esta extração vai percorrer o método passado por argumento, e todos aqueles que esse método utiliza, e vai gerar um ficheiro com uma variável global *Leakages*. Esta variável *Leakages* é utilizada como um acumulador de dados e informação que pode ser vista por um intruso.

Para provar tal propriedade, depois de extrair o programa para *EasyCrypt* vamos criar um novo ficheiro onde vamos especificar que se passarmos ao mesmo programa diferentes valores de *input*, vamos obter o mesmo resultado de *Leakages* do *EasyCrypt*. Se a variável *Leakages* for igual nas duas execuções percebemos que estas não dependem dos segredos ou dos *inputs*, o que nos leva a concluir que é um algoritmo segura contra ataques de *constant-time*.

Numa primeira tentativa de prova de *constant-time* percebemos que seria necessário algum trabalho extra sobre este programa. Idealmente apenas teríamos de invocar os procedimentos gerados automaticamente duas vezes, indicando que os argumentos em cada um deles teriam os mesmos apontadores, e que queríamos provar que os dois obteriam o mesmo resultado de *Leakages*. No exemplo em baixo veremos um esboço de como seria fazer uma prova deste tipo, tendo em conta que o ficheiro da implementação em *Jasmin* já teria sido extraído para *EasyCrypt*:

```
require import Blake2b_init_fullstatefull_CT.

equiv blake2b_init_ct :
       M.blake2b_init_jazz ~ M.blake2b_init_jazz
           := {ctxt, outlen, key, keylen} ==> ={M.leakages}.
proof. proc;inline *;sim => />. qed.
```

Como a nossa implementação do algoritmo *Blake2b* suporta uma *API*, e exige um contexto específico que guarda o estado a cada chamada de qualquer função desta interface, seria necessário uma investigação mais profunda.

Devido à existência de uma variável que representa todo o contexto, desde contadores, *buffers*, etc, a forma como exprimimos *Easycrypt* para provar a propriedade de *constant-time* seria diferente pois dentro do contexto temos zonas de memória que seriam consideradas públicas (como o por exemplo o *outlen*) e outras que necessariamente teriam de ser privadas (como o *buffer* e o estado interno). Por isso não bastaria passar apenas esse apontador, teríamos de arranjar forma de informar o programa que variáveis do contexto é que nós queríamos assumir como segredo ou aquelas que seriam irrelevantes.

Para o prazo deste projeto esta investigação não seria praticável, mas deixamos uma base para uma investigação futura.

7

ANÁLISE DE RESULTADOS

Neste capítulo vamos fazer testes de *performance* sobre os algoritmos implementados e vamos analisar os resultados obtidos. Visto que o nosso contributo passa pela implementação do algoritmo *Blake2b* ,vamos analisar apenas a eficiência desse algoritmo, visto que o *Argon2* apenas usa esta função a nível interno e também existe uma enorme quantidade de instruções extras. Assim será mais fácil perceber o impacto das nossas codificações se analisarmos ao nível da própria função de *hash* e não na função de derivação de chaves.

Estes testes passam por medir o tempo de execução. Para tal vamos tirar partido da biblioteca $< time.h >$ da linguagem C e vamos usar o método $clock()$ que devolve o número de *ticks* executados pelo computador desde o início do programa. Para medirmos o tempo real de duração do algoritmo vamos invocar este método $clock()$ imediantamente antes e imediatamente depois da execução da função de *hash*. Depois subtraímos os dois valores, para obtermos o número total de *ticks* durante o decorrer do algoritmo e dividimos pelo número de ciclos de *CPU* por minuto, obtendo assim a duração, em segundos, do programa. Para melhor análise vamos apresentar o resultado em *ms*. Para um resultado mais fidedigno este processo de medir o tempo de duração vai ser repetido 1000 vezes sendo devolvido a média da duração.

Como vimos pelos capítulos anteriores uma função de *hash* criptográfica consegue receber um *input* de tamanho arbitrário e transformá-lo num *output* de tamanho fixo. Também foi referido anteriormente que a versão do algoritmo *Blake2* a ser implementada seria a versão *Blake2b* que permite devolver *outputs* até 64 *bytes* (512 *bits*). Esta versão também permite ao utilizador usar chaves para calcular um *hash*. Chave esta que também tem um tamanho máximo de 64 bytes.

Posto isto, para o *Blake2b* vamos usar um cenário de teste onde o tamanho do vetor de saída é de 32 *bytes*, visto que é um tamanho muito comum nas funções de *hash*, e uma chave de 64 *bytes*.

7.1 BLAKE2B

Como vimos no capítulo 4 foram feitas cinco implementações diferentes: a implementação de referência, as duas escalares a versão com interface e a otimizada com instruções vetorizadas. Decidimos excluir as duas versões escalares desta análise pois estas foram apenas versões intermédias entre a versão de referência e aquela que seria a versão que suportaria a interface. Tendo em conta que o algoritmo tem implementações ofi-

ciais na linguagem *C* vamos comparar as nossas versões em *Jasmin* com as versões de referência e otimizada em *C*, de forma a percebermos se a nossa implementação consegue ser mais eficiente.

Antes da análise dos resultados vamos descrever os resultados que esperamos obter.

Como foi referido o *Jasmin* é uma ferramenta que permite um melhor controlo e gestão da memória, e o facto de ter semelhanças ao C e o compilador estar preparado para produzir código *Assembly* eficiente, é de esperar que as versões do algoritmo em *Jasmin* sejam, na pior das hipóteses, tão boa como a versão de referência na linguagem *C*.

Quanto às diferentes versões de *Jasmin* a nossa versão de referência será a menos eficiente entre elas, seguindo-se da versão com interface e depois a otimizada.

Por último ficamos com as versões otimizadas nas duas linguagens. Como vimos em cima neste documento, a versão otimizada em *C* utiliza técnicas de vectorização o que lhe dá, teoricamente, uma vantagem considerável perante todas as outras versões. A nossa versão otimizada em *Jasmin* inspirou-se nessa mesma versão em *C* e espera-se que tenha uma *performance* muito semelhante.

Passaremos agora à análise da *performance* entre as diferentes versões.

7.1.1 *Todas as versões*

Vamos então comparar todas as versões do algoritmo com um só gráfico.

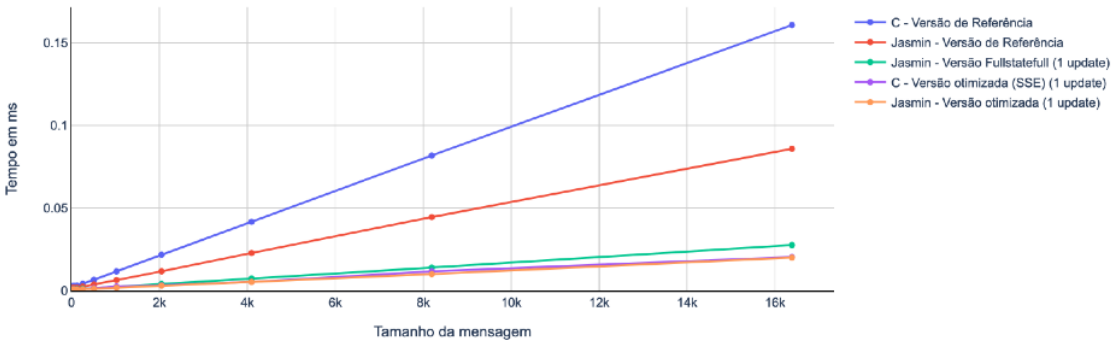

Figure 12: Tempo de execução de todas as versões

Num panorama geral vemos que os resultados vão de encontro ao esperado. Vamos nas subsecções seguintes analisar mais ao pormenor isolando apenas algumas versões para obtermos uma melhor análise.

7.1.2 *Implementações de Referência*

As implementações de referência são as implementações menos eficientes. Este resultado era o esperado visto que a única preocupação era implementar um código funcional sem pensar em possíveis melhorias de *performance*.

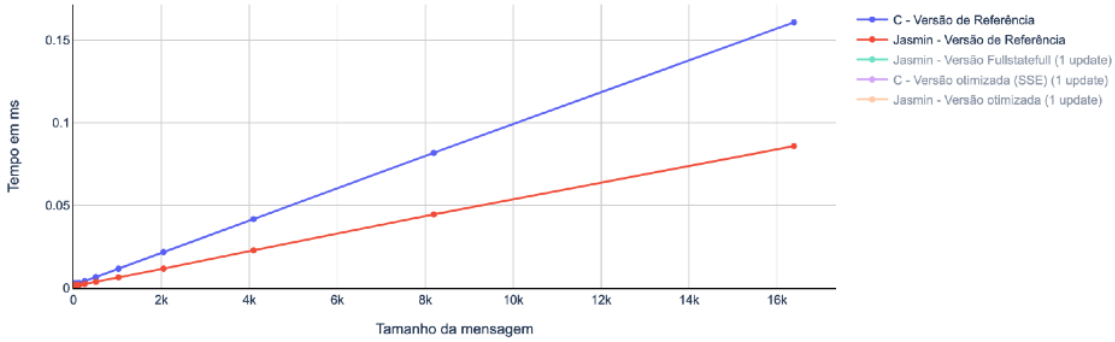

Figure 13: Tempo de execução das versões de referência

Ainda assim vemos uma melhoria significativa da nossa versão comparando à versão de referência da linguagem C.

7.1.3 *Implementações fullstatefull e o otimizadas*

Resta-nos então analisar as versões *fullstatefull* e as otimizadas.

Pelo gráfico vemos que também vai de encontro ao resultado que esperávamos. Apesar da melhoria entre a versão *fullstatefull* e a versão de referência, ambas em *Jasmin*, ser bastante significativa, a versão otimizada em *C* acaba por ser bem mais eficiente. É um resultado expectável visto que as instruções vetorizadas da versão otimizada, dá-lhe um grande avanço perante a nossa versão *fullstatefull*. Como dissemos anteriormente, também seria esperado que as duas versões tivessem um resultado muito semelhante porque ambas usam instruções vetorizadas, e os ganhos de uma linguagem em relação à outra, deverão estar, entre outras razões, no custo de *overhead* em cada uma delas.

Vamos agora isolar as três versões para obtermos um tamanho mais apropriado:

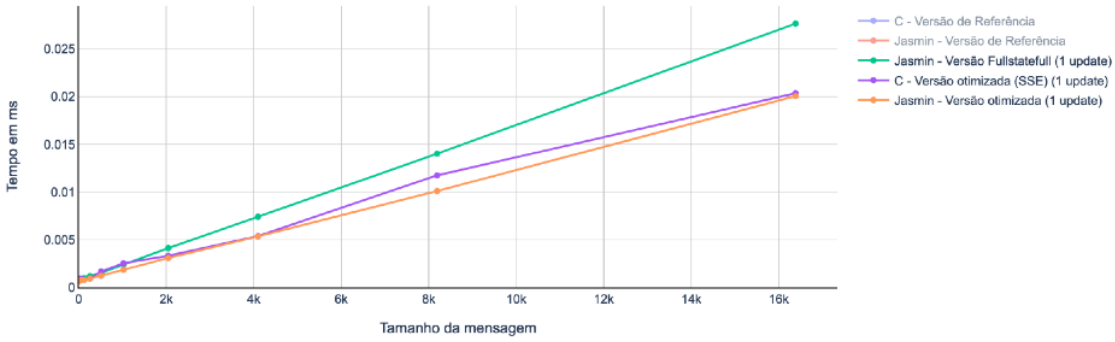

Figure 14: Tempo de execução das versões *fullstatefull* e otimizadas

Como vemos pela imagem acima, as versões otimizadas obtiveram um ganho substancial, sendo mais eficientes que a versão *fullstatefull*.

Quanto às duas versões otimizadas vemos que a versão em *Jasmin* será ligeiramente mais eficiente que a versão em *C*, o que nos deixa bastante satisfeitos com esta implementação.

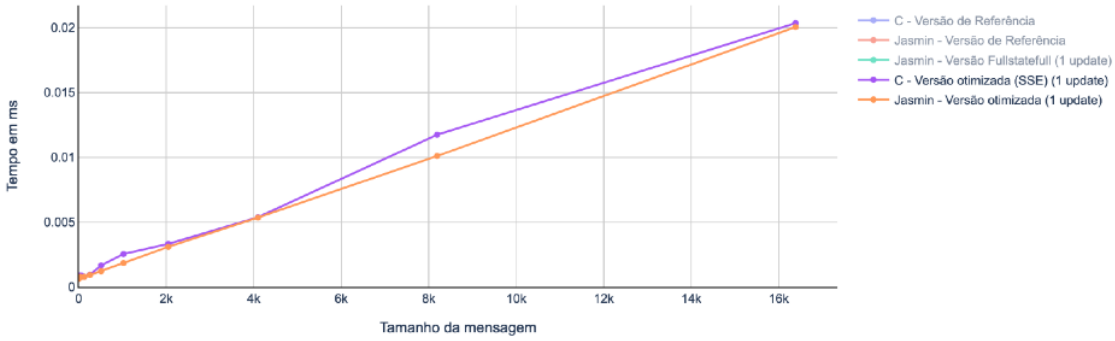

Figure 15: Tempo de execução das versões otimizadas

8

CONCLUSÃO

Durante este projeto estudamos o conceito de gestores de *passswords* dedicamos tempo a perceber como alguns funcionam, desde rotinas e procedimentos executados no alto nível, até ao mais baixo nível como os algoritmos criptográficos utilizados. De todos eles o *KeePass* destacou-se devido ao número de utilizadores que eles apresentam e pelo facto de ser um software *open-source*, o que nos permitiu investigar mais a fundo e verificar que os seus algoritmos são atuais e ainda considerado seguros .

O facto de atuarmos sobre os algoritmos em si permitiu que o nosso contributo pudesse ser utilizado a nível global e não só para um gestor em específico, isto é, qualquer software que utilizasse a função de *hash Blake2b*, ou ainda o *Argon2*, apesar do nossos foco ter sido apenas na função de *hash*, pudesse pegar na nossa codificação em *Jasmin* e adaptasse a sua interface em qualquer linguagem, desde que pudesse invocar o *Assembly* do nosso programa gerado pelo compilador do *Jasmin*. Além disso como esta linguagem é ainda recente e muito direcionada para a criptografia, pudemos também contribuir para a sua comunidade apresentando uma solução inovadora no que diz respeito à *API* adotada e suportada. A partir da nossa implementação mostramos ser possível implementar interfaces típicas desta área e que esta tecnologia tem o potencial necessário para construirmos muitas mais primitivas criptográficas e algoritmos. Neste momento contribuímos com uma nova função de *hash*, já bastante otimizada e eficiente, e deixamos um protótipo que poderá servir como base para a prova de *constant-time* do algoritmo.

BIBLIOGRAPHY

[1] Google, Harris Poll. Online Security Survey https://services.google.com/fh/files/blogs/google_security_infographic.pdf

[2] Malwarebytes. Password Manager. Disponível em: https://www.malwarebytes.com/what-is-password-manager/. Acedido em 6 de outubro às 16:46

[3] Página oficial do Jasmin. Disponível em: https://jasmin-lang.github.io

[4] EasyCrypt. Disponível em: https://www.easycrypt.info/trac/

[5] Página oficial do KeePass. Disponível em: https://keepass.info/

[6] Página oficial do Keepass2Android. Disponível em: https://play.google.com/store/apps/details?id=keepass2android.keepass2android&hl=pt_PT&gl=US

[7] Página oficial do gestor de passwords Google Chrome Password Manager. Disponível em: https://passwords.google.com/

[8] Gordon Kelly. Google Has 5 Exciting Upgrades For Google Chrome Browser Users. Disponível em: https://www.forbes.com/sites/gordonkelly/2020/08/02/google-chrome-performance-security-feature-payment-windows-10-android-ios-?sh=521cfbbe73a6#131aa17c73a6 . Acedido em 18 de Dezembro de 2020 às 17:57

[9] Página oficial do gestor de passwords Firefox Lockwise. Disponível em: https://www.mozilla.org/pt-PT/firefox/lockwise/

[10] Página oficial do gestor de passwords 1Password. Disponível em: https://1password.com/

[11] Password Managers: Under the Hood of Secrets Management. Disponível em https://www.ise.io/casestudies/password-manager-hacking/

[12] Página oficial do gestor de passwords LastPass. Disponível em: https://www.lastpass.com/pt/

[13] Key derivation functions. Disponível em: https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions.html

[14] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and provably repairing the ssh au-thenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM*

Trans. Inf. Syst. Secur., 7(2):206–241, may 2004. ISSN 1094-9224. doi: 10.1145/996943.996945. URL https://doi.org/10.1145/996943.996945.

[15] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1807–1823, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi:10.1145/3133956.3134078. URL https://doi.org/10.1145/3133956.3134078.

[16] Página oficial do Algoritmo Blake2. Disponível em: https://www.blake2.net/

[17] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: Simpler, smaller, fast as md5. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security*, pages 119–135, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38980-1.

[18] Markku-Juhani Saarinen and Jean-Philippe Aumasson. The *BLAKE2 Cryptographic Hash and Message Authentication Code (MAC): IETF RFC 7693*. Number 7693 in Request for Comments. Internet Engineering TaskForce, nov 2015. doi: 10.17487/RFC7693.

[19] Repositório oficial com as implementações do Blake2. Disponível em: https://github.com/BLAKE2/BLAKE2/tree/master/ref

[20] Página oficial da biblioteca Cryptography do Python. Disponível em: https://cryptography.io/en/latest/

[21] Página oficial da biblioteca Web Crypto do JavaScript. Disponível em https://www.w3.org/TR/WebCryptoAPI

[22] Página oficial da biblioteca Node.Js Crypto. Disponível em: https://nodejs.org/api/crypto.html

[23] Página oficial da Intel Intrinsics Guide. Disponível em: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

[24] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. pages 292–302, 03 2016. doi: 10.1109/EuroSP.2016.31.

[25] A. Biryukov, D. Dinu, D. Khovratovich, S. Josefsson. The memory-hard Argon2 password hash and proof-of-work function draft-irtf-cfrg-argon2-13. 2021. URL: https://tools.ietf.org/pdf/draft-irtf-cfrg-argon2-13.pdf

[26] Repositório oficial com as implementações do Argon2. Disponível em https://github.com/P-H-C/phc-winner-argon2