Hash Criptográfico sobre Senhas e Aleatoriedade do Argon2*

Pietro Di Consolo Gregorio[1], **Denise Hideko Goya**[1]

[1]Centro de Matemática, Computação e Cognição – Universidade Federal do ABC
Avenida dos Estados, 5001, Santo André, SP, CEP 09210-580

pietro.gregorio@aluno.ufabc.edu.br, denise.goya@ufabc.edu.br

Abstract. *The Argon2 algorithm, elected in the Password Hashing Competition, has three distinct variants indicated for specific scenarios. Since the version chosen by the user cannot be guaranteed, this paper discusses about the three versions and analyzes the behavior of the hashes generated with different parameters in respect to two randomness metrics: entropy and monobit. As a result, Argon2i and Argon2d presented results as expected in the scenarios for which they were planned, but Argon2id did not.*

Resumo. *O algoritmo Argon2, eleito na Password Hashing Competition, possui três versões distintas indicadas para uso em cenários específicos. Como não se pode garantir que a versão ideal será selecionada pelo fabricante ou pelo usuário final, este trabalho analisa o comportamento dessas versões com diferentes parâmetros, com relação a duas métricas de aleatoriedade: entropia e monobit. Como resultados, confirma-se que as versões Argon2i e Argon2d apresentam resultados de acordo com os esperados nos cenários para os quais foram planejadas, mas a versão híbrida Argon2id não.*

1. Introdução

Muitas soluções criptográficas dependem de uma função *hash* para garantirem suas propriedades de segurança. O intuito de uma função *hash* é mapear um dado de comprimento indefinido, através de uma série de transformações matemáticas, a um dado de comprimento definido [Menezes et al. 2001]. Uma função *hash* criptográfica considerada segura deve apresentar determinadas propriedades como, por exemplo, dado um valor de *hash* qualquer, é computacionalmente inviável encontrar uma entrada que produza tal *hash*, propriedade chamada de resistência à pré-imagem; dada uma entrada qualquer, é computacionalmente inviável encontrar outra entrada, diferente da primeira, que produza o mesmo valor de *hash*, o que é chamado de resistência à segunda pré-imagem; e é computacionalmente inviável encontrar um par distinto que produza o mesmo valor de *hash*, que é conhecido como resistência a colisão [Stavroulakis and Stamp 2010]. Tais propriedades foram comprometidas em funções como a SHA-0 [Wang et al. 2005], MD5 [Sotirov et al. 2009] e SHA-1 [Stevens et al. 2017], o que torna os algoritmos, protocolos e sistemas baseados nestas funções inseguros.

Um dos usos de funções *hash* é em *Key Derivation Function* (KDF), cujo objetivo é gerar chaves criptográficas a partir de um valor dado como entrada. Quando a entrada da KDF é uma senha acrescida de um número aleatório exclusivo para cada usuário, chamado de sal (ou *salt*), tem-se uma solução mais adequada para se aumentar a entropia

*Este trabalho recebeu apoio financeiro da UFABC e do CNPq.

da senha [Wagner et al. 1998], já que as senhas criadas por usuários tendem a ter uma entropia baixa [NIST 2013]. Essa prática ajuda na prevenção de ataques de dicionário [NIST 2009], caracterizados por tentativas de adivinhação de senhas com base em uma relação de palavras contidas em um dicionário [Bellare et al. 2000]. Funções KDF aplicadas sobre senha com sal também são conhecidas por *Password Hashing Scheme* (PHS).

Entre 2013 e 2015, ocorreu a *Password Hashing Competition* (PHC), uma competição aberta para selecionar um PHS mais seguro, pois os disponíveis até então apresentavam estruturas similares ao do SHA-1, considerado inseguro. Também eram vulneráveis a ataques de canais laterais (*side-channel attacks*) [Ladeira et al. 2016, Ventura and Dahab 2009] e tentativas de adivinhação de senha por força-bruta, usando plataformas de processamento paralelo, como GPU e FPGA [Dürmuth and Kranz 2015]. A competição elegeu o PHS chamado Argon2 [Biryukov et al. 2017] como seu ganhador e fez menção honrosa a mais quatro finalistas: Lyra2, Catena, Makwa e yescrypt.

Há três versões de Argon2: o Argon2d, mais rápido e com acesso dependente de dados à memória, indicado para a utilização em casos sem ameaça de ataque de temporização (*timing attack*); o Argon2i, que usa um acesso independente de dados à memória e é mais lento por fazer mais passos para se proteger de ataques de balanceamento entre memória e processamento (*tradeoff attacks*); e o Argon2id, um híbrido das duas primeiras versões, que possui parte da resistência a ataques de canais laterais (*side-channel attacks*) da versão 2i e da resistência a ataques de GPU da versão 2d [Biryukov et al. 2017]. Uma vez que a indústria paulatinamente está aderindo à recomendação de substituir os algoritmos PHS mais antigos pelo Argon2 e, no entanto, não é evidente quais são as reais condições em que cada versão de Argon2 deve ser adotada, em especial a híbrida, justifica-se um estudo mais detalhado para explanar o funcionamento deste algoritmo (Seção 2) e avaliar sua segurança através de métricas de aleatoriedade selecionadas (Seção 3).

2. Argon2

Argon2 é um esquema de *hashing* de senhas baseado em funções conhecidas como *memory-hard functions*, visando possuir a maior taxa de preenchimento de memória junto com uma proteção contra ataques de *tradeoff*, explorando a organização do cache e da memória dos processadores mais recentes da Intel e AMD.

O Argon2 possui duas principais versões, o Argon2d e o Argon2i, além de uma função adicional híbrida dos dois primeiros, o Argon2id [Biryukov et al. 2017]. A principal diferença entre estas funções está na função de indexação (*indexing function*) usada. Pelo seu funcionamento, o Argon2d é adequado para criptomoedas, por exemplo, e o Argon2i é indicado para o *hashing* de senhas e derivação de chaves baseada em senha.

2.1. Produto tempo-área e memory-hard functions

Um dos objetivos do Argon2 é maximizar o custo da quebra de senha em ASIC's (*Application Specific Integrated Circuit*). Para os autores, a forma mais apropriada de medir o custo foi através do produto tempo-área [Thompson 1979, apud Biryukov et al. 2017].

Para produzir o valor de *hash* são empregados um certo tempo por senha e alguns núcleos da CPU, usando uma quantidade M de memória, que por sua vez, pode ser relacionada à área A do ASIC. O tempo de funcionamento do ASIC é denotado por T.

Aumentar o custo então seria o equivalente a maximizar o produto AT. Para diminuir a memória de um ASIC (o que equivale a diminuir sua área) usa-se um fator $\alpha < 1$ para obtermos uma diminuição αM. Será necessário, usando um *tradeoff*, gastar $C(\alpha)$ vezes o mesmo cálculo e o tempo de funcionamento aumentará no mínimo por um fator $D(\alpha)$ e portanto, o máximo ganho $\mathcal{E}_{max}$ no produto tempo-área será: $\mathcal{E}_{max} = \max_\alpha \frac{1}{\alpha D(\alpha)}$

Caso $D(\alpha) > 1/\alpha$ quando $\alpha \to 0$, a função *hash* é chamada *memory-hard* e o produto tempo-área não diminui.

A matriz da memória $B[\,]$ é preenchida com uma função de compressão G:

$B[0] = H(P, S),\ \ B[j] = G\big(B[\phi_1(j)], B[\phi_2(j)], \cdots, B[\phi_t(j)]\big),$ para j de 1 a t

Onde $\phi_i(\,)$ são funções de indexação.

Pode-se dividir as funções de indexação usadas em dois tipos [Biryukov et al. 2017]:

- Independente da senha e do sal, mas podendo ser dependente de outros parâmetros públicos. Os endereços de memória podem ser calculados pelo adversário, podendo ter acesso paralelo e pré-armazenar dados, além de, caso o atacante use um ataque *tradeoff* de tempo-espaço, os blocos faltantes podem ser pré-calculados. Se um único *core* G ocupa uma porção β da memória total, então, se for usado αM de memória, o ganho no produto tempo-área será de: $\mathcal{E}(\alpha) = \frac{1}{\alpha + C(\alpha)\beta}$
- Dependente da senha, impede que sejam pré-calculados e adquiridos dados necessários antecipadamente, fazendo com que o atacante só precise calcular quando for necessário. Se o dado for calculado como uma árvore de F chamadas de profundidade D, então o ganho será: $\mathcal{E}(\alpha) = \frac{1}{(\alpha + C(\alpha)\beta)D(\alpha)}$

Além disso, o tempo de execução de um ataque é limitado pela largura de banda máxima do equipamento adversário.

2.2. O funcionamento do Argon2

Pode-se dividir as entradas do Argon2 entre primárias (a senha P e o sal S) e secundárias (outros parâmetros). Cada entrada possui um formato esperado: P deve ter um tamanho de 0 a $(2^{32} - 1)$ bytes; S deve ter tamanho de 8 a $(2^{32} - 1)$; o grau de paralelismo p, que determina quantos cálculos independentes e sincronizados serão permitidos, deve ser um valor inteiro de 1 a $(2^{24} - 1)$; o tamanho τ da *Tag* (*hash* final), deve ser um inteiro de 4 a $(2^{32} - 1)$ bytes; o tamanho m de memória, deve ser um número inteiro de kilobytes de $8p$ a $(2^{32} - 1)$, sendo o número efetivo de blocos, m', igual a m arredondado para baixo para o valor mais próximo de um múltiplo de $4p$; o número de iterações t deve ser qualquer inteiro de 1 a $(2^{32} - 1)$; a versão v é um byte $0x13$; o valor secreto K, que pode ser usado como chave, mas no padrão não é usado, deve ter qualquer tamanho entre 0 e $(2^{32} - 1)$ bytes; dados associados X, com qualquer tamanho entre 0 e $(2^{32} - 1)$ bytes; a variante y do Argon2 (0 para Argon2d, 1 para Argon2i, 2 para Argon2id).

A função de compressão G recebe duas entradas de 1024 bytes e gera uma saída de 1024 bytes. A função *hash* H interna do Argon2 é o Blake2b.

Sem paralelismo (Figura 1), o Argon2 funciona da seguinte forma: a função G é iterada m vezes, no passo i um bloco de índice $\phi(i) < i$ é tomado da memória ($\phi(i)$ é determinado pelo bloco anterior no Argon2d ou é um valor fixo no Argon2i). O conceito

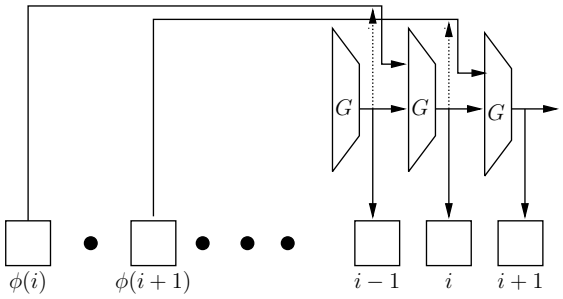

Figura 1. Funcionamento do Argon2 sem paralelismo [Biryukov et al. 2017].

de *extract-then-expand* (extrair e então expandir) [Krawczyk 2008], é seguido pelo Argon2: é extraída a entropia da senha e do sal aplicando-se uma função *hash* sobre eles. Os parâmetros e o tamanho das entradas P, S, K, X são também colocados como entrada.

$$H_0 = H(p, \tau, m, t, v, y, \langle P \rangle, P, \langle S \rangle, S, \langle K \rangle, K, \langle X \rangle, X)$$

Com H_0 sendo um valor de 64 bytes e os parâmetros $p, \tau, m, t, v, y, \langle P \rangle, \langle S \rangle, \langle K \rangle, \langle X \rangle$ são tratados como inteiros de 32 bits, com a extremidade menor primeiro (*little-endian*).

A memória é preenchida com $m' = \lfloor \frac{m}{4p} \rfloor \cdot 4p$ blocos de 1024 bytes, organizada em uma matriz $B[i][j]$, com p linhas e $q = m'/p$ colunas. Os blocos produzidos pelo passo t são representados como $B^t[i][j], t > 0$, e são calculados da seguinte forma:

$$B^1[i][0] = H'(H_0 || \underbrace{0}_{\text{4 bytes}} || \underbrace{i}_{\text{4 bytes}}), \quad 0 \leq i < p;$$
$$B^1[i][1] = H'(H_0 || \underbrace{1}_{\text{4 bytes}} || \underbrace{i}_{\text{4 bytes}}), \quad 0 \leq i < p;$$
$$B^1[i][j] = G(B^1[i][j-1], B^1[i'][j']), \quad 0 \leq i < p,\ 2 \leq j < q.$$

Na qual G é a função de compressão, H' é uma função *hash* de tamanho variável construída com H e o índice $[i'][j']$ é determinado de forma diferente para o Argon2d e Argon2i. Caso $t > 1$, em vez de sobrepor os blocos antigos pelos novos, é aplicada uma operação lógica de XOR (ou exclusivo) para escrever o bloco novo na memória. Ao fazer de T iterações na memória, é calculado o bloco final B_{final} como o XOR da última coluna:

$$B_{\text{final}} = B^T[0][q-1] \oplus B^T[1][q-1] \oplus \cdots \oplus B^T[p-1][q-1].$$

E então é aplicado H' em B_{final} para gerar a *Tag*, ou o valor de *hash* final. A função *hash* usada como base para criar a função *hash* de tamanho variável é o Blake2b, que suporta saídas de 1 a 64 bytes. Para tornar possível saídas maiores do que 64, é usado o Blake2b diversas vezes, sendo a primeira entrada o conjunto tamanho e dado, e as diversas outras vezes com a saída anterior. Assim, a *Tag* é o conjunto dos primeiros 32 bytes (*little-endian*) de cada saída da função, formando a *hash* final anexando-se estas saídas uma atrás da outra.

2.3. Implementações do Argon2

As versões do Argon2 possuem fraquezas que delimitam as possíveis aplicações da cada uma. O Argon2d, que usa uma função de indexação dependente, não sobrescreve a memória, limitando seu uso em cenários com ataques de coleta de "resíduo", ou seja, se o atacante possui acesso direto à máquina poderá adquirir dados do processo feito pelo algoritmo que o ajudará a descobrir a entrada.

O Argon2d é otimizado para cenários no qual o possível atacante não possua acesso à memória ou à CPU, portanto, é indicando para casos em que não se pode realizar ataques de temporização por canal-lateral e nem ataques de coleta de resíduo, como citado anteriormente. Os cenários típicos indicados pelos autores é autenticação em servidores *backend* e mineração de criptomoedas.

O Argon2i, com função de indexação independente, é indicado para cenários mais perigosos pois estão mais expostos a atacantes, sendo que estes podem ter acesso à mesma máquina, usando sua CPU e possuindo acesso direto. Diferentemente do Argon2d, o Argon2i sobrescreve duas vezes a memória em apenas três passagens, sendo indicado para derivação de chave (*key derivation*) e também para autenticação em servidores *frontend*.

As especificações da versão 1.3 do Argon2 [Biryukov et al. 2017] usada neste artigo, indicam um procedimento para a escolha do algoritmo. O primeiro passo é a escolha da versão, apontando que o Argon2i seja adotado caso não saiba a diferença entre as versões ou caso ataques de canal-lateral possam acontecer. Os outros passos são: máximos de *threads* que podem ser iniciadas em cada chamada do Argon2; máxima memória e tempo para cada chamada; tamanho do sal e da *Tag*; descobrir o máximo de passos que podem ser feitos sem que passem o tempo dado anteriormente. Estes parâmetros devem ser passados para o Argon2 na sua utilização.

3. Métricas de Aleatoriedade

Para avaliar a *hash* gerada pelo Argon2, foram usadas a entropia e frequência (monobit). A entropia é uma medida matemática da quantidade de incerteza sobre mensagens geradas por uma fonte [Shannon 1948]; a noção de alta entropia, ou seja, uma maior incerteza sobre o dado, é usada na criptografia moderna para a preservação de sigilo, integridade de dados, bem como para garantir a autenticidade das partes envolvidas numa comunicação [Denning 1982, NIST 2013].

Seguindo as métricas recomendadas pelo NIST para avaliar a aleatoriedade da saída, foi escolhido também a frequêcia monobit que, ao verificar a proporção de zeros e uns em uma sequência, permite concluir se a aleatoriedade é suficiente para aplicações seguras através do valor de um parâmetro P, para o qual valores menores que 0,01 indicam que uma sequência não é aleatória [NIST 2010].

A senha usada para os testes foi "Alice123" e o sal "UFABC2018". Para cada versão do Argon2 (i, d e id) foram geradas *hashes* variando-se apenas o parâmetro a ser observado e deixando os outros parâmetros no padrão da implementação oficial.

Assim, para cada versão, foram formados quatro conjuntos de *hashes*, o primeiro conjunto foi obtido com a variação do número de iterações, coletando-se todas as saídas geradas, de 1 a 1000 iterações. No segundo conjunto variou-se o valor de uso da memória de 2^3 a 2^{23} KiB. Já no terceiro conjunto o paralelismo foi avaliado de 1 a 500 *threads*. Por

fim, no quarto conjunto o tamanho da *hash* final foi variado de 4 até 100 bytes. Após a coleta de cada conjunto foram calculados os valores de entropia e os valores de frequência das *Tags*, avaliando o comportamento do algoritmo para as diferentes variações e versões.

3.1. Resultados para Entropia

Como os PHS geram *hashes* fazendo operações com a entrada, quanto menos um atacante souber previamente sobre a saída, melhor será o algoritmo. Portanto esta característica desejada justifica um estudo sobre a entropia do Argon2, que representa a incerteza sobre uma determinada informação desconhecida, avaliando-se a entropia e as diferenças entre cada versão, investigando se este seria um fator influente na escolha.

Os gráficos apresentados nas Figuras 2, 3 e 4 para as versões i, d e id respectivamente, apresentam os valores de entropia calculados para os as *hashes* do primeiro conjunto, no qual o número de iterações foi variado, sendo cada iteração uma passagem pelo algoritmo, isto é, para iteração igual a dois, por exemplo, a senha passa pelo Argon2 e depois a *Tag* gerada passa novamente pelo Argon2, gerando a *hash* final.

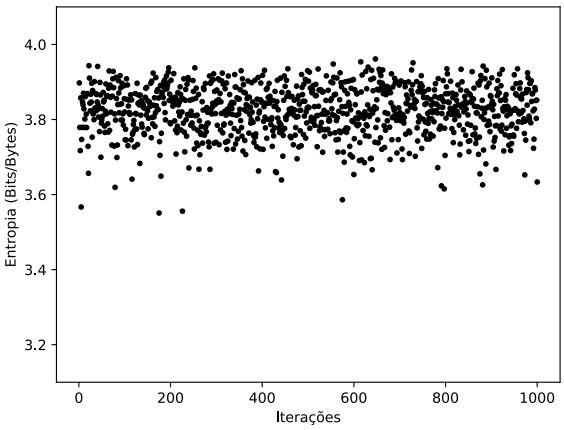

Figura 2. Entropia *vs* Iterações para o Argon2i.

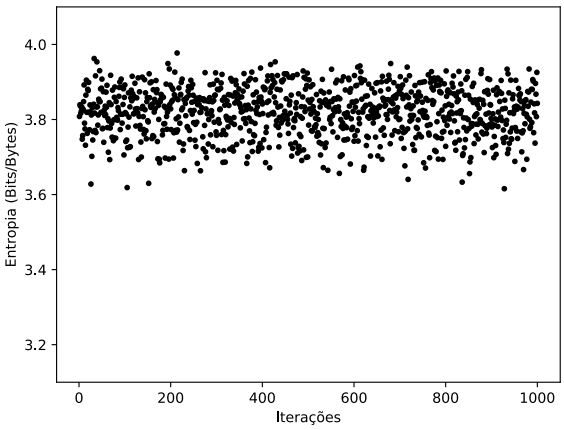

Figura 3. Entropia *vs* Iterações para o Argon2d.

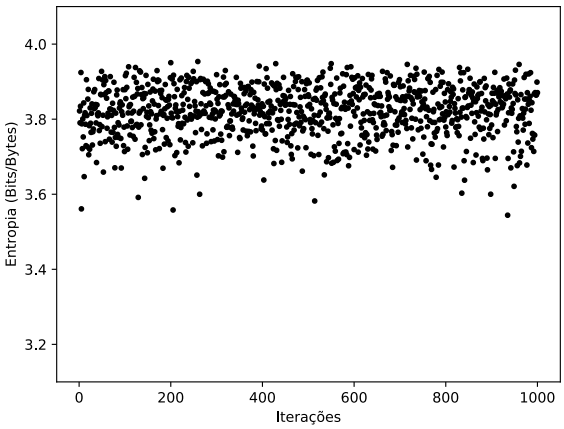

Figura 4. Entropia *vs* Iterações para o Argon2id.

Pode-se notar que os valores de entropia flutuam em um certo intervalo. O intervalo de variação da entropia foi ligeiramente mais estreito para o Argon2d, mas a entropia média foi ligeiramente maior no Argon2i, conforme se observa na coluna "Iterações" da Tabela 1. Ambos resultados são coerentes com o propósito das versões 2i e 2d, pois:

- entropia mais alta significa maior incerteza e dificulta o mapeamento de relações entre a saída e a entrada (que é a senha, cuja independência com a memória se pretendeu garantir com a versão 2i);
- menor quantidade de casos cuja entropia se distanciou da maior parte das execuções significa uma distribuição mais homogênea e maior dificuldade em relacionar entradas e saídas sem acesso a estados de memória por um atacante (propósito da versão 2d).

Usando os valores calculados de entropia para a variação de uso de memória e de paralelismo, os gráficos ficam semelhantes ao da variação de iterações, observando-se o mesmo comportamento: a entropia varia dentro de uma faixa.

A média dos valores de entropia nestes casos fornece o valor central da faixa na qual a entropia está variando. Os valores médios de entropia, com suas respectivas incertezas, para cada versão e para cada conjunto são apresentados na Tabela 1.

Tabela 1. Entropia média de cada conjunto para cada versão.

	Entropia (bit/byte)		
Versão	**Iterações**	**Memória**	**Paralelismo**
Argon2i	(3,823 $\pm$ 0,002)	(3,82 $\pm$ 0,01)	(3,821 $\pm$ 0,003)
Argon2d	(3,820 $\pm$ 0,002)	(3,83 $\pm$ 0,01)	(3,821 $\pm$ 0,003)
Argon2id	(3,820 $\pm$ 0,002)	(3,82 $\pm$ 0,01)	(3,818 $\pm$ 0,003)

Os valores de entropia não variam muito entre cada versão do Argon2, assim como não há discrepâncias entre os valores dos diferentes conjuntos. Isso indica que para as variações de iterações, memória e paralelismo, com os valores padrões dos outros parâmetros, a entropia das *Tags* possui valores médios bem próximos.

Como visto nos gráficos, a entropia não apresenta uma dependência substancial com o número de iterações, nem com o uso da memória e o paralelismo, podendo-se afirmar que a entropia é aleatória dentro de uma faixa de valores. Este intervalo é o que torna o Argon2 um bom PHS: tanto com poucas iterações como com muitas, os valores de entropia permanecem contidos no intervalo, que possui um valor de entropia tendendo ao valor médio, que é próximo de quatro bits por byte.

No entanto, era de se esperar que a versão Argon2id apresentasse resultados intermediários entre 2d e 2i em todos os testes e isso não ocorreu. Conforme se observa na Tabela 1, o Argon2id apresentou valores de entropia sempre mais baixos nos três testes e se igualou ao Argon2d com relação ao teste de Iterações; se igualou ao Argon2i com relação ao teste de Memória e foi o pior em Paralelismo. Se considerarmos que o usuário indeciso sobre qual das três versões usar tem boa chance de optar pela versão intermediária 2id caso não leia a documentação com atenção, também terá boa chance de não fazer a melhor escolha. O impacto dessa possível má escolha merece ser investigado em detalhes em trabalhos futuros.

Os gráficos das Figuras 5, 6 e 7 mostram que a relação entre a entropia e o tamanho da *hash* é diferente da relação com os outros parâmetros.

Como citado anteriormente, o tamanho da *hash* foi variado entre 4 e 100 bytes. Para tamanhos de até 20 bytes, a entropia cresce de valor rapidamente, já que a entropia

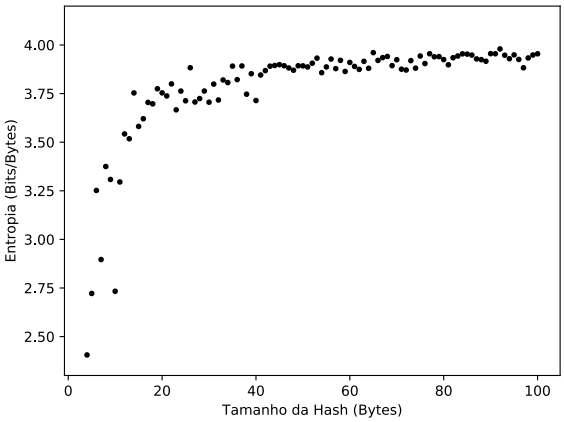

Figura 5. Entropia *vs* Tamanho da *Hash* para o Argon2i.

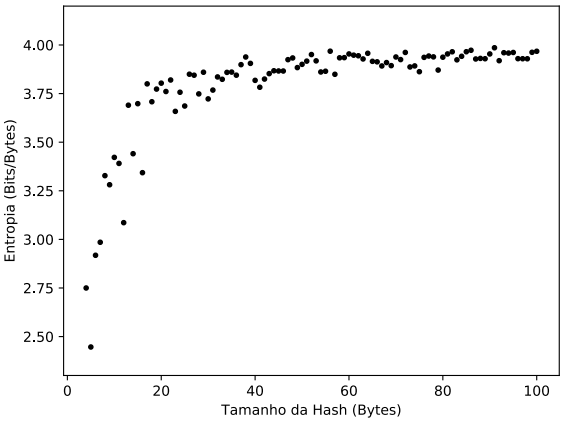

Figura 6. Entropia *vs* Tamanho da *Hash* para o Argon2d.

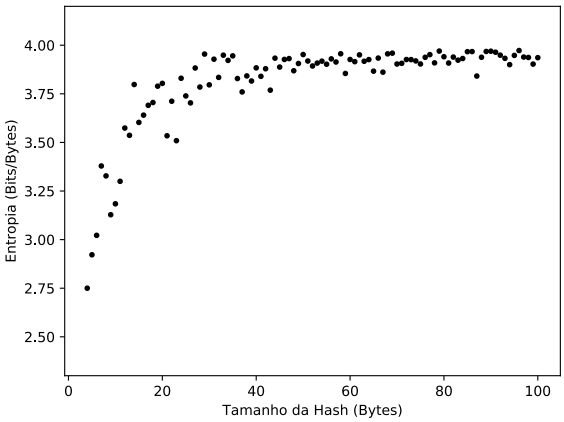

Figura 7. Entropia *vs* Tamanho da *Hash* para o Argon2id.

está relacionada por uma função logarítmica com o tamanho da saída. Após 20 bytes, a entropia muda de valor lentamente, tendendo à 4 bits por byte.

Esse resultado mostra que, do ponto de vista da entropia, valores mais altos que 20 bytes devem ser adotados para não ter problemas de segurança. Essa vulnerabilidade se dá também caso o problema seja visto da perspectiva de que um PHS mapeia entradas às saídas: quanto menor a saída, menor vai ser o espaço de saídas possíveis, sendo assim maior a chance de se encontrar diferentes entradas com colisões em tempos menores.

Um dos pontos importantes é que a implementação oficial usa como padrão *hashes* de 32 bytes, que é um valor já na parte mais estável do gráfico. Uma indicação de que tamanhos abaixo de 20 bytes não devem ser usados poderia ser dada na documentação oficial para orientar sobre este parâmetro.

3.2. Resultados para Frequência (Monobit)

Para todos os *hashes* coletados, foram calculados o valor de frequência para avaliar se a saída pode ser considerada aleatória ou não, sendo que quanto mais saídas fossem consideradas aleatórias melhor seria, pois seria menos previsível. A Tabela 2 apresenta, em porcentagem, quantos *hashes* foram considerados não aleatórios.

Pode-se notar que a quantidade de *hashes* consideradas não aleatórias é baixa para todos os tipos de de variações, sugerindo que o Argon2 não possui fraqueza na

Tabela 2. Porcentagem de *hashes* não aleatórias pelo método monobit.

	Iterações	**Memória**	**Paralelismo**	**Tam. da *hash***
Argon2i	0,6 %	0,0 %	1,0 %	2 %
Argon2d	0,7 %	4,8 %	0,4%	0 %
Argon2id	0,8 %	0,0 %	0,4 %	1 %

aleatoriedade das *Tags*.

O Argon2id mostrou-se um meio termo entre os valores para as variações de memória, paralelismo e tamanho da *hash*, mas apresentou mais números não aleatórios na variação de iterações. Portanto, é interessante se realizar mais testes para verificar se a escolha do número de iterações no Argon2id apresenta limiares entre níveis de segurança esperados e indesejados.

4. Conclusão

Pelas avaliações feitas neste trabalho sobre *hashes*, o Argon2 atende aos requisitos de um PHS moderno, mas possui três versões distintas cujos cenários de uso devem ser bem compreendidos pelos usuários antes de sua escolha.

A aplicação de cada versão é indicada pelos criadores do Argon2, destacando que o Argon2i deve ser usado para o *hashing* de senha, pois sobrescreve a memória rapidamente evitando ataques de coleta de resíduo e é mais lento, evitando ataques de *tradeoff*. O Argon2d é indicado para cenários nos quais o atacante não possua acesso direto, não podendo assim fazer ataques de canal lateral de temporização e nem de coleta de resíduo. Já o Argon2id deve ser usado em cenários similares ao do Argon2d no qual algumas das características do 2i é desejada, mas não deve ser usado em casos em que o 2i é indicado, pois as características do 2d podem comprometer a segurança, portanto o Argon2i deve ser adotado para casos que não se sabe os riscos.

As características avaliadas apontaram diferenças pequenas entre as três versões e que os comportamentos para o Argon2d e Argon2i foram como os esperados. Os parâmetros usados como padrão do Argon2 possuem entropia e frequências do tipo monobit satisfatórias, o que é muito importante pois o PHS poderá ser implementado de forma direta e segura, sem a necessidade de alterar estes valores caso não se saiba calculá-los.

No entanto, foi detectado que a versão que apontou comportamento mais incerto foi o Argon2id, pois se igualou ao 2d ou 2i, dependendo do caso, e foi o pior no teste de entropia sobre o parâmetro paralelismo, além de também ter sido o pior no teste de aleatoriedade por monobit e iterações. Isso indica que o usuário deve compreender muito bem as razões para escolher o Argon2id, em vez do 2d ou 2i. Uma leitura desatenta à documentação pode sugerir ao usuário que a versão 2id é uma opção intermediária e que se aplica a quaisquer cenários mistos entre os indicados para 2d e 2i.

Sugere-se, portanto, como trabalhos futuros, uma investigação mais detalhada sobre possíveis impactos do uso inadequado da versão Argon2id, nos casos em que o ideal seria adotar as versões 2d ou 2i. Também é interessante estudar outras métricas, além da aleatoriedade monobit e entropia, que possam reforçar os resultados discutidos sobre o Argon2id.

Referências

Bellare, M., Pointcheval, D., and Rogaway, P. (2000). Authenticated key exchange secure against dictionary attacks. *EUROCRYPT'00*, pages 139–155.

Biryukov, A., Dinu, D., and Khovratovich, D. (2017). Argon2: the memory-hard function for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302, Luxemburgo.

Denning, D. E. R. (1982). *Cryptography and Data Security*. Addison-Wesley, EUA.

Dürmuth, M. and Kranz, T. (2015). On password guessing with GPUs and FPGAs. *Technology and Practice of Passwords: International Conference on Passwords (PASSWORDS'14)*, pages 19–38.

Krawczyk, H. (2008). On Extract-then-Expand key derivation functions and an HMAC-based KDF.

Ladeira, L. Z., Nascimento, E. N., Ventura, J. P. F., Dahab, R., Aranha, D. F., and Hernández, J. C. L. (2016). Canais laterais em criptografia simétrica e de curvas elípticas: ataques e contramedidas. *SBSeg*, 16:82–141.

Menezes, A., van Oorschot, P., and Vanstone, S. (2001). *Handbook of Applied Cryptography*. CRC Press, Estados Unidos da América.

NIST (2009). Special publication 800-108 - recommendation for key derivation using pseudorandom functions. *NIST*.

NIST (2010). Special Publication 800-22 Rev. 1a. A statistical test suite for random and pseudorandom number generators for cryptographic applications. *NIST*.

NIST (2013). Special publication 800-63-2 - electronic authentication guideline. *NIST*.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656.

Sotirov, A., Stevens, M., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D. A., and de Weger, B. (2009). Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. *CRYPTO 2009*, pages 55–69.

Stavroulakis, P. and Stamp, M. (2010). *Handbook of Information and Communication Security*. Springer, Alemanha.

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Markov, Y. (2017). The first collision for full SHA-1. Cryptology ePrint Archive 2017/190.

Thompson, C. D. (1979). Area-time complexity for VLSI. In *ACM Symposium on Theory of Computing*, STOC '79, EUA. ACM.

Ventura, J. P. F. and Dahab, R. (2009). Introdução a ataques por canais secundários. *SBSeg*, 9:3–47.

Wagner, D., Kelsey, J., Schneier, B., and Hall, C. (1998). Secure applications of low-entropy keys. *Information Security Workshop (ISW'97)*, pages 121–134.

Wang, X., Yu, H., and Yin, Y. L. (2005). Efficient collision search attacks on SHA-0. *CRYPTO 2005*, pages 1–16.