DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico Final

**pbcrypt: optimización de password crackers con SSE4 y AVX2**

Organización del Computador II
2019



| Integrante | LU | Correo electrónico |
|---|---|---|
| Gonzalo Juarros, Catalina | 673/15 | catalinajuarros@gmail.com |

**Resumen**

Las instrucciones SIMD son especialmente útiles para algoritmos en los que no hay muchas dependencias a nivel datos. Los algoritmos de cifrado por bloques utilizados para proteger la privacidad de las contraseñas, como es el caso de bcrypt, no tienen esta característica. Sin embargo, en un tipo de aplicación llamado *password cracker*, la presencia de grandes volúmenes de datos independientes entre sí abre oportunidades de vectorización que no existen cuando un algoritmo criptográfico procesa una sola entrada. Se diseñó una variante de bcrypt que opera con registros de 128 bits y bloques de cuatro claves en simultáneo, a la que se denominó pbcrypt, y una que maneja registros de 256 bits y bloques de ocho claves, llamada pbcrypt doble. Se escribieron programas en lenguaje ensamblador x86-64 con extensiones SSE4 y AVX2 para estos dos algoritmos, así como para otras variantes de bcrypt con variaciones en algunos detalles implementativos, y se comparó su rendimiento con el de una implementación basada en instrucciones de propósito general. Se obtuvo una mejora del 33 % en la tasa de *hashes* por segundo con la variante de cuatro claves y una del 175 % con la de ocho claves.

# Índice

**1. Introducción** **1**

1.1. Alcance de este trabajo . . . . . 1

1.2. Especificaciones técnicas . . . . . 1

**2. Algoritmos** **2**

2.1. Nociones preliminares . . . . . 2

2.2. Blowfish . . . . . 2

2.2.1. El cifrado por bloques . . . . . 2

2.2.2. Pseudocódigo . . . . . 3

2.3. bcrypt . . . . . 3

2.3.1. Pseudocódigo . . . . . 4

2.4. *Password crackers* . . . . . 5

**3. Paralelización** **6**

3.1. Pseudocódigo . . . . . 9

3.2. 8 contraseñas y generalización . . . . . 11

**4. Implementación** **12**

4.1. Estructura . . . . . 12

4.2. bcrypt: versión escalar . . . . . 12

4.3. bcrypt: versión vectorizada . . . . . 14

4.3.1. pbcrypt doble . . . . . 15

4.4. Optimizaciones . . . . . 16

4.4.1. *Loop unrolling* . . . . . 16

4.4.2. Subclaves en registros YMM . . . . . 17

4.5. *Cracker* . . . . . 18

4.5.1. Argumentos y precondiciones . . . . . 18

4.5.2. bcrypt . . . . . 19

4.5.3. pbcrypt . . . . . 19

4.5.4. Aclaración sobre las versiones . . . . . 19

4.6. *Tests* . . . . . 20

4.7. Cifrado con bcrypt . . . . . 20

**5. Experimentos** **21**

5.1. Métodos y condiciones de los experimentos . . . . . 21
5.1.1. Especificaciones del sistema . . . . . 21
5.1.2. Generación de datos de entrada . . . . . 21
5.1.3. Ejecución de experimentos . . . . . 22
5.2. Complejidad temporal, *wordlist* y contraseña . . . . . 22
5.2.1. Tamaño de la *wordlist* . . . . . 22
5.2.2. Longitud de la contraseña . . . . . 24
5.3. Complejidad temporal, rondas . . . . . 24
5.4. Impacto del tamaño del *batch* en el rendimiento . . . . . 25
5.5. Alineamiento de código a 64 bytes . . . . . 26
5.6. Penalización por transiciones AVX-SSE . . . . . 28
5.7. Comparación con la implementación de OpenBSD . . . . . 29
5.8. Comparación entre instrucciones SIMD y accesos a memoria . . . . . 31
5.9. pbcrypt sin la instrucción más costosa . . . . . 32
5.10. pbcrypt doble . . . . . 34
5.10.1. *Wordlist* creciente . . . . . 34
5.10.2. Costo . . . . . 35

**6. Conclusiones** **37**

**A. Especificaciones del sistema**

# 1. Introducción

Las instrucciones de tipo **vectorial** o **SIMD** (*Single Instruction Multiple Data*) brindan la posibilidad de operar con muchos datos en simultáneo. Esto se debe a que manejan registros anchos que pueden almacenar dos o más valores al mismo tiempo. Su uso se presta muy directamente para **optimizar el rendimiento** de algoritmos en los que hay **pocas o nulas dependencias** entre los datos procesados; uno de los ejemplos más habituales –visto durante la materia para la que se realizó este trabajo– es realizar las mismas operaciones sobre varios pixels de una imagen en paralelo para aplicarle un filtro.

Por otra parte, las **funciones criptográficas** suelen tener muchas **interdependencias** al nivel de los datos; es frecuente que éstos se procesen secuencialmente como bloques contiguos donde el resultado de operar sobre uno es necesario para el cómputo con el siguiente, que a su vez es requerido para todos los posteriores. Más adelante se tratarán los motivos de esta característica, pero en principio lo importante es remarcar que debido a ella hay **muy pocas posibilidades de vectorizar** esta clase de algoritmos.

En este trabajo se investiga una técnica para usar instrucciones vectoriales en el algoritmo criptográfico **bcrypt**, más específicamente en el contexto de un ***password cracker***. Cuando se encripta una sola contraseña, las dependencias entre sus bloques de datos son un impedimento para optimizar la implementación con operaciones SIMD; sin embargo, en un *cracker* es necesario cifrar muchas contraseñas distintas y cada encriptación es **completamente independiente** de todas las demás, lo cual abre **nuevas oportunidades de vectorización**.

Aunque la motivación principal es analizar el impacto de procesar bloques de varias contraseñas en simultáneo con instrucciones SIMD sobre el rendimiento del *password cracker*, también se exploran otras optimizaciones posibles detalladas en secciones posteriores.

## 1.1. Alcance de este trabajo

La investigación aquí presentada se enfoca en los temas más pertinentes al programa de Organización del Computador II, en particular a algoritmos vectorizados (el segundo trabajo práctico de la cursada) y algunos aspectos de optimización de código ASM. Aunque se realiza una comparación con la implementación C de referencia, no se pretende ahondar en esta cuestión ni en los interrogantes que surgen de ella. Tampoco se tratan en profundidad aspectos más relacionados con ingeniería de software de los *password crackers*, como la incorporación de *multithreading* o técnicas de generación de contraseñas.

## 1.2. Especificaciones técnicas

Este trabajo se llevó a cabo con un procesador de arquitectura **Intel x86-64** con extensiones **SSE4** y **AVX2**. La elección de este hardware se debe a que x86 fue la arquitectura estudiada en Organización del Computador II y se contaba con fácil acceso a un procesador de estas características.

A lo largo de este informe, el término *dword* designa un valor de 32 bits (4 bytes). Los nombres de algunos parámetros y las palabras extranjeras que no estén muy naturalizadas ni sean nombres propios de algoritmos aparecen en *itálica*, y los nombres de archivos y porciones de código fuente aparecen en `monoespaciada`. Dado que bcrypt es una función de *hash* criptográfica, los términos *hash* y texto cifrado se usan de manera indistinta; lo mismo ocurre con hashear y encriptar o cifrar.

# 2. Algoritmos

## 2.1. Nociones preliminares

Se asume conocimiento esencial previo acerca de funciones de encriptación y de *hash*, particularmente qué es un cifrado por bloques; en caso de considerar que no se tiene suficiente familiaridad con estos temas, se recomienda consultar *Introduction to Modern Cryptography*, de Katz y Lindell [5]. Si el lector ya se siente cómodo con Blowfish, bcrypt y/o el funcionamiento de un *password cracker*, puede saltear esta sección e ir directamente a la 3.

La primera subsección describirá la función criptográfica **Blowfish**, ya que bcrypt se construye a partir de ella. La mayor parte de los aspectos algorítmicos que se analizan en este trabajo, de hecho, son propios de Blowfish, mientras que los principales aportes de bcrypt son incrementar la dificultad de crackeo y pasar de una función de cifrado reversible a una de *hash*.

## 2.2. Blowfish

Este algoritmo criptográfico fue desarrollado por Bruce Schneier en 1993 [15]. Toma como parámetros los datos a cifrar y una clave de entre 32 y 448 bits. Se basa en una estructura denominada **contexto** (`blf_ctx` en el código), compuesta por:

- un arreglo bidimensional de enteros de 32 bits, de 4 por 256 elementos, llamado $S$ (del inglés *state*, estado), donde a cada sub-arreglo se lo llama *S-box*
- un arreglo unidimensional de 18 enteros de 32 bits llamado $P$

Inicialmente, los elementos de $S$ y $P$ contienen los dígitos hexadecimales de $\pi$. Lo primero que hace Blowfish es encriptar $P$ con la clave a través de un **cifrado XOR**, repitiendo bytes de la clave de ser necesario. En el paso siguiente ya empieza a haber dependencias entre los datos, pues comienza el **cifrado por bloques**: cada par de elementos consecutivos de $P$ –o sea, cada bloque de 64 bits– se encripta con **16 rondas de una función** $F$ (en la próxima subsección se profundiza sobre el funcionamiento del cifrado por bloques y por qué introduce tantas dependencias). Una vez encriptado $P$, se realiza el mismo procedimiento de cifrado por bloques con $S$, caja por caja. Todos los pasos recién descriptos componen una rutina llamada ***key setup***. Se recomienda entender en detalle dicha rutina (ver algoritmo 1), debido a que es un **ladrillo fundamental de la estructura de bcrypt**. El contexto final se usa para encriptar datos de longitud arbitraria, que en el caso de bcrypt consisten en un string a partir del que se obtiene el eventual valor del *hash*.

### 2.2.1. El cifrado por bloques

El método de encriptación que emplea Blowfish pertenece a una clase de cifrados por bloques conocidos como **cifrados de Feistel**. Para una explicación más detallada, se recomienda consultar *Handbook of Applied Cryptography* [10], pero básicamente consisten en ejecutar un cierto número de **rondas** sobre un **bloque de datos**. El bloque se entiende como dos mitades, inicialmente $L_0$ y $R_0$, y cada $i$-ésima ronda, para $i \geq 1$, se compone de estos pasos:

- $L_i \leftarrow R_{i-1}$
- $R_i \leftarrow L_{i-1} \oplus F(R_{i-1}, K_i)$, donde $K$ es la clave de encriptación y $K_i$ es alguna **subclave** obtenida a partir de ella

La función $F$, los valores concretos de $L_0$ y $R_0$ y la forma de derivar cada $K_i$ a partir de $K$ varían según el algoritmo de cifrado particular, pero la estructura básica es la que se acaba de exponer

(en Blowfish, los bloques miden 64 bits, $L_0$ y $R_0$ son ambos 0 y las $K_i$ se obtienen del contexto; $F$ es engorrosa de describir y es más sencillo dar su pseudocódigo).

El hecho de que $L_{i+1}R_{i+1}$ dependa **directamente** de $L_iR_i$, que a su vez depende de $L_{i-1}R_{i-1}$, implica que, sea cual sea el algoritmo de cifrado, **es imposible paralelizar el cómputo de rondas**. Como se verá en el pseudocódigo, encriptar cada bloque de 64 bits del contexto potencialmente requiere del resultado de encriptar otro, de lo cual se deduce directamente que **el cifrado de los bloques de un mismo contexto no se puede vectorizar**.

En conjunción con otras decisiones de diseño, estas dependencias, generalmente hablando, derivan en características deseables para un algoritmo de cifrado, pues ocultan patrones en los datos y así protegen contra ciertos métodos de criptoanálisis [11].

### 2.2.2. Pseudocódigo

**Algorithm 1** Blowfish

```
1:  function BLOWFISH(context, K)                    ▷ El contexto debe estar inicializado con π
2:      S ← context.S
3:      P ← context.P
4:      for 0 ≤ i < 18 do
5:          P_i ← P_i ⊕ K_{i mod |K|}
6:      (L, R) ← (0, 0)
7:      for 0 ≤ i < 18; i ← i + 2 do
8:          ENCIPHER(L, R, S, P)
9:          (P_i, P_{i+1}) ← (L, R)
10:     for 0 ≤ i < 4 do
11:         for 0 ≤ j < 256; j ← j + 2 do
12:             ENCIPHER(L, R, S, P)
13:             (S_{i,j}, S_{i,j+1}) ← (L, R)

14: function ENCIPHER(L, R, S, P)
15:     for 0 ≤ i < 16 do
16:         L ← L ⊕ P_i
17:         R ← F(L, S) ⊕ R
18:         (L, R) ← (R, L)
19:     (L, R) ← (L ⊕ P_16, R ⊕ P_17)
20:     (L, R) ← (R, L)

21: function F(X, S)
22:     (B0, B1, B2, B3) ← (X_0, X_1, X_2, X_3)      ▷ bytes de X
23:     return (S_{0,B0} + S_{1,B1}) ⊕ S_{2,B2} + S_{3,B3}
```

En la línea 5, si $|K|$ mód $4 \neq 0$ (siendo $|K|$ la longitud de la clave en bytes), se vuelven a contar los bytes desde el principio para armar el bloque de 4.

## 2.3. bcrypt

Mientras que Blowfish es un algoritmo de encriptación reversible, bcrypt es una función de *hash*. Fue diseñada por Niels Provos y David Mazières en 1999 [7] y es el algoritmo de *hash* de contraseñas por defecto del sistema operativo OpenBSD [8]. bcrypt aprovecha el **costo computacional** del *key setup* de Blowfish, que es alto en comparación con el de la mayoría de los algoritmos de cifrado, para resistir ataques de fuerza bruta. Para este trabajo, se tomó como referencia la implementación escrita por Provos y Ted Unangst para OpenBSD.

bcrypt acepta contraseñas de **hasta 72 bytes**. En líneas generales, consiste en cifrar repetidamente un **arreglo de 24 bytes**, inicializado con el *string* `"OrpheanBeholderScryDoubt"`, usando como clave de encriptación la contraseña dada. El resultado de este cómputo es el ***hash* de la contraseña**. Para incrementar todavía más su resistencia a ataques, introduce dos nuevos parámetros; uno es un valor de **128 bits** denominado **sal** y el otro es $\boldsymbol{C}$, el **costo**: el logaritmo en base 2 de la cantidad de veces que se lleva a cabo el *key setup*. El propósito de la sal será explicado en la subsección sobre *password crackers*. El costo se introdujo para parametrizar la dificultad computacional de bcrypt de forma que no sea necesario diseñar un nuevo algoritmo frente a los avances en el hardware y el subsiguiente aumento en la potencia de los *crackers* [9].

### 2.3.1. Pseudocódigo

**Algorithm 2** bcrypt

**function** BCRYPT($K, salt, C$) ▷ $K$ es la clave, $salt$ la sal, $C$ el costo
    hash ← `"OrpheanBeholderScryDoubt"`
    context ← InitialContext
    BLOWFISHEXPAND(context, $K$, $salt$)
    **for** $0 \leq i < 2^C$ **do**
        BLOWFISH(context, $K$)
        BLOWFISH(context, $salt$)
    **for** $0 \leq i < 64$ **do**
        BLOWFISHENCRYPT(hash, context)
    **return** hash

**function** BLOWFISHEXPAND($context, K, salt$)
    $S \leftarrow context.S$
    $P \leftarrow context.P$
    **for** $0 \leq i < 18$ **do**
        $P_i \leftarrow P_i \oplus K_{i \bmod |K|}$
    $(L, R) \leftarrow (0, 0)$
    **for** $0 \leq i < 18; i \leftarrow i + 2$ **do**
        $(L, R) \leftarrow (L \oplus salt_{i \bmod |salt|}, R \oplus salt_{(i+1) \bmod |salt|})$
        ENCIPHER($L, R, S, P$)
        $(P_i, P_{i+1}) \leftarrow (L, R)$
    **for** $0 \leq i < 4$ **do**
        **for** $0 \leq j < 256; j \leftarrow j + 2$ **do**
            $(L, R) \leftarrow (L \oplus salt_{i \bmod |salt|}, R \oplus salt_{(i+1) \bmod |salt|})$
            ENCIPHER($L, R, S, P$)
            $(S_{i,j}, S_{i,j+1}) \leftarrow (L, R)$

**function** BLOWFISHENCRYPT($hash, context$)
    $S \leftarrow context.S$
    $P \leftarrow context.P$
    **for** $0 \leq i < |hash|; i \leftarrow i + 2$ **do** ▷ $i$ cuenta bloques de 32 bits en vez de bytes
        ENCIPHER($hash_i, hash_{i+1}, S, P$)

BLOWFISH y ENCIPHER son sencillamente las funciones definidas en el pseudocódigo 1. BLOWFISHEXPAND es similar a BLOWFISH, pero al cifrado de Feistel le suma un cifrado XOR con la sal. BLOWFISHENCRYPT es la función que se usaría en Blowfish para cifrar datos cualesquiera, que en este caso fue aprovechada para generar un *hash* de 24 bytes.

## 2.4. *Password crackers*

En seguridad informática, se le llama *password cracker* a una aplicación cuyo objetivo es, dada una contraseña encriptada o *hasheada*, **recuperar su valor en texto plano**. Si dicha contraseña fue cifrada con un algoritmo criptográficamente fuerte, no debería ser posible para el *cracker* realizar ningún tipo de criptoanálisis, por lo que la única forma que tendría de hallar el texto plano original sería probando **muchas contraseñas** por **fuerza bruta**.

Dado que probar todas las combinaciones posibles de caracteres, incluso acotando la longitud, es extremadamente poco práctico en términos de tiempo, los *password crackers* generalmente usan ***wordlists***, es decir, **archivos de texto** compuestos por valores que se consideran más probables como contraseña original. Pueden ser, por ejemplo, listas de **palabras comunes** en ciertos idiomas, o listas de **contraseñas reales** en texto plano obtenidas ilegítimamente de alguna base de datos (como es el caso de `rockyou.txt`) [2]. Algunos *crackers* también generan nuevas contraseñas candidatas a partir de reglas de mutación –el lector puede referirse a [6] para más detalles–, pero esto último se escapa del alcance de este trabajo.

Teniendo una *wordlist*, el *password cracker* puede calcular el *hash* o texto cifrado de cada contraseña hasta encontrar alguno que coincida con la entrada, lo cual se conoce como **ataque de diccionario**. Es ahora donde se vuelve muy relevante la **sal** mencionada en la subsección 2.3. Este valor, que en un esquema criptográfico bien implementado se obtiene de manera aleatoria, es un número de $t$ bits que se usa como entrada del algoritmo de cifrado con el objetivo de dificultar este tipo de ataques. Si un atacante tiene que **hallar la sal además de la contraseña**, deberá calcular $2^t$ textos cifrados para cada elemento de la *wordlist* [12], por lo que el **espacio de búsqueda** es **mucho mayor** y el costo de efectuar un ataque se vuelve **prohibitivo** para cierto modelo de adversario. Si la sal **se guarda junto con la contraseña cifrada** (que de hecho es lo que se hace en OpenBSD), obviamente no se cuenta con esta garantía contra los ataques de diccionario más ingenuos, pero sí contra una **optimización** a éstos conocida como ***rainbow tables***: bases de datos que recopilan los valores de *hash* para contraseñas conocidas [13]. Si se piensa la *rainbow table* como un diccionario, la clave de la consulta es el *hash* y el valor devuelto es la contraseña en texto plano de la que éste se obtuvo. Cuando el texto cifrado que busca el atacante está en la *rainbow table*, obtener la contraseña original sólo toma **una consulta**, por más fuerte que sea el algoritmo criptográfico usado. Sin embargo, si hay que usar una sal para la encriptación, hay hasta $2^t$ resultados distintos para cada contraseña, por lo que el tamaño de la base de datos necesaria aumenta en un factor de $2^t$ y ahora es su **almacenamiento** lo que se torna un **obstáculo** para el atacante. Dada esta situación, debería llevar a cabo un ataque de diccionario ingenuo, que por supuesto es mucho más laborioso que uno con *rainbow tables*.

# 3. Paralelización

Tal como se vio en el apartado 2.2.1, las dependencias de datos de los algoritmos de cifrado por bloques los vuelven malos candidatos para la optimización con instrucciones vectoriales en lo que respecta a encriptar una única entrada. Sin embargo, los *password crackers* operan con **varias contraseñas independientes entre sí**, lo cual introduce la posibilidad de **procesamiento paralelo a nivel de datos**. En esta sección se describirán en detalle las adaptaciones a Blowfish y bcrypt que permiten **hashear más de una contraseña al mismo tiempo** con **instrucciones SIMD**. Antes de exhibir la versión vectorizada de bcrypt, se presentarán algunas operaciones en orden ascendente de complicación, con el objetivo de facilitar la comprensión del algoritmo y de las decisiones de diseño detrás de éste. Se denominará **pbcrypt** (por *parallel bcrypt*) al algoritmo vectorizado.

La idea básica es operar de forma análoga a la exhibida en los algoritmos 1 y 2, pero usando registros SIMD que contengan **bloques de más de una entrada en simultáneo**. Así, por ejemplo, la operación

$$L \leftarrow L \oplus P_i \tag{1}$$

se traduce a

$$(L^0 L^1 L^2 L^3) \leftarrow (L^0 L^1 L^2 L^3) \oplus (P_i^0 P_i^1 P_i^2 P_i^3) \tag{2}$$

donde cada $L^m$ corresponde al cifrado de la $m$-ésima contraseña ($0 \leq m \leq 3$) y cada $P_i^m$ se obtuvo del contexto correspondiente a ésta. Particularmente, en una arquitectura de la familia x86, esta operación se efectúa con un registro XMM de 128 bits (pues los $L^m$ y $P_i^m$ miden 32 bits cada uno) y la instrucción `pxor` (o `vpxor`).

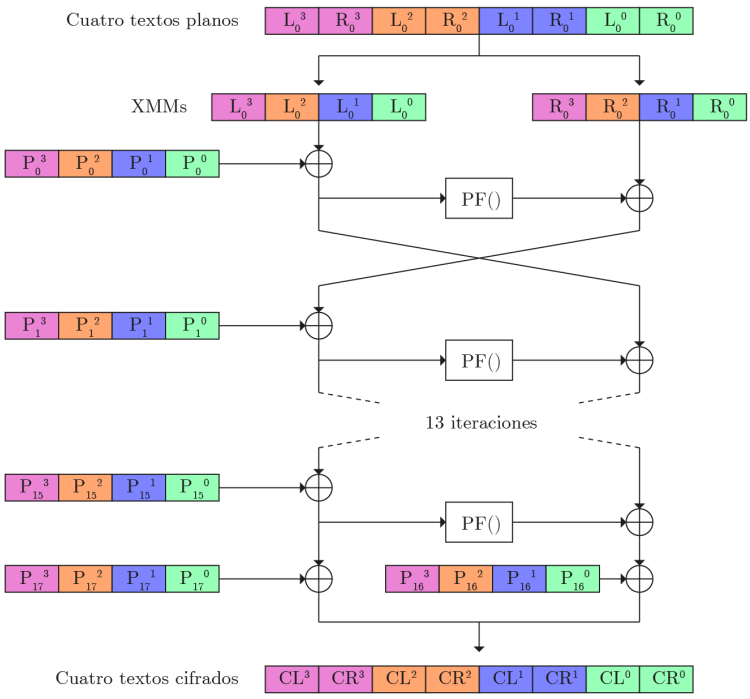


Figura 1: Cifrado paralelo en 16 rondas. Diagrama adaptado de [1].

Es muy importante notar que, en el ejemplo anterior, el subíndice $i$ es el mismo para todos los $P_i^m$. Esto es porque $i$ indica qué **bloque de datos** del **contexto correspondiente** a la $m$-ésima contraseña se está usando para el cifrado. Si cada $L^m$ se inicializa en 0, tal como $L$ en los algoritmos 1 y 2, necesariamente se tiene que llevar a cabo el mismo número de ronda de encriptación para cada contraseña; como en la $i$-ésima ronda se usa siempre $P_i$, se deduce directamente que el elemento con el que se cifra cada $L^m$ debe tener el **mismo índice** en el arreglo $P$ que le corresponde.

Para que el cifrado paralelo se haga de manera eficiente, es conveniente poder leer todos los $P_i^m$ con **un único acceso a memoria**, en vez de leer cada uno de un contexto correspondiente a una contraseña distinta. Esto motiva el diseño de una estructura de datos que se denominará **contexto paralelo** (`p_blf_ctx` en el código). Dicha estructura es similar al contexto descripto en la sección 2.2, pero con la diferencia fundamental de que tanto el arreglo $P$ como las *S-boxes* tienen **cuatro veces más elementos**; para evitar ambigüedades, se llamará $S'$ y $P'$ a los arreglos del contexto paralelo. Empieza con cuatro copias contiguas en memoria de cada elemento del contexto inicial, en el mismo orden. Los siguientes ejemplos, así como la figura 2, ilustran el patrón:

- $S'_{0,0}$, $S'_{0,1}$, $S'_{0,2}$ y $S'_{0,3}$ son iguales a $S_{0,0}$
- $S'_{0,4}$, $S'_{0,5}$, $S'_{0,6}$ y $S'_{0,7}$ son iguales a $S_{0,1}$
- $S'_{1,0}$, $S'_{1,1}$, $S'_{1,2}$ y $S'_{1,3}$ son iguales a $S_{1,0}$
- $P'_0$, $P'_1$, $P'_2$ y $P'_3$ son iguales a $P_0$

| | | $K^0$ | $K^1$ | $K^2$ | $K^3$ | $K^0$ | $K^1$ | $K^2$ | $K^3$ |
|---|---|---|---|---|---|---|---|---|---|
| **S[0]** | [0] | d1310ba6 | d1310ba6 | d1310ba6 | d1310ba6 | 98dfb5ac | 98dfb5ac | 98dfb5ac | 98dfb5ac |
| | [8] | 2ffd72db | 2ffd72db | 2ffd72db | 2ffd72db | d01adfb7 | d01adfb7 | d01adfb7 | d01adfb7 |
| | | ... | | | | | | | |
| | [1016] | 08ba4799 | 08ba4799 | 08ba4799 | 08ba4799 | 6e85076a | 6e85076a | 6e85076a | 6e85076a |
| **S[1]** | [0] | 4b7a70e9 | 4b7a70e9 | 4b7a70e9 | 4b7a70e9 | b5b32944 | b5b32944 | b5b32944 | b5b32944 |
| | [8] | db75092e | db75092e | db75092e | db75092e | c4192623 | c4192623 | c4192623 | c4192623 |
| | | ... | | | | | | | |
| | [1016] | e6e39f2b | e6e39f2b | e6e39f2b | e6e39f2b | db83adf7 | db83adf7 | db83adf7 | db83adf7 |
| **S[2]** | [0] | e93d5a68 | e93d5a68 | e93d5a68 | e93d5a68 | 948140f7 | 948140f7 | 948140f7 | 948140f7 |
| | [8] | f64c261c | f64c261c | f64c261c | f64c261c | 94692934 | 94692934 | 94692934 | 94692934 |
| | | ... | | | | | | | |
| | [1016] | 670efa8e | 670efa8e | 670efa8e | 670efa8e | 406000e0 | 406000e0 | 406000e0 | 406000e0 |
| **S[3]** | [0] | 3a39ce37 | 3a39ce37 | 3a39ce37 | 3a39ce37 | d3faf5cf | d3faf5cf | d3faf5cf | d3faf5cf |
| | [8] | abc27737 | abc27737 | abc27737 | abc27737 | 5ac52d1b | 5ac52d1b | 5ac52d1b | 5ac52d1b |
| | | ... | | | | | | | |
| | [1016] | 578fdfe3 | 578fdfe3 | 578fdfe3 | 578fdfe3 | 3ac372e6 | 3ac372e6 | 3ac372e6 | 3ac372e6 |
| **P** | [0] | 243f6a88 | 243f6a88 | 243f6a88 | 243f6a88 | 85a308d3 | 85a308d3 | 85a308d3 | 85a308d3 |
| | [8] | 13198a2e | 13198a2e | 13198a2e | 13198a2e | 03707344 | 03707344 | 03707344 | 03707344 |
| | | ... | | | | | | | |
| | [64] | 9216d5d9 | 9216d5d9 | 9216d5d9 | 9216d5d9 | 8979fb1b | 8979fb1b | 8979fb1b | 8979fb1b |

Figura 2: Contexto paralelizado correspondiente a los contextos de las claves $K^0$, $K^1$, $K^2$ y $K^3$. Estado inicial.

Así, una lectura a memoria de 128 bits alcanza para obtener los $P_i^m$ de la operación 2.

Dado que un contexto paralelo representa los contextos de cuatro contraseñas distintas, la primera parte de la adaptación para pbcrypt de las funciones Blowfish y BlowfishExpand –el cifrado XOR con el parámetro $K$– también debe hacerse con varias entradas. En vez de leer cuatro bytes de una única clave para cifrar $P_i$, deben leerse **cuatro bytes de cada contraseña** en un mismo registro de 16 bytes para cifrar $P'_{4i}$, $P'_{4i+1}$, $P'_{4i+2}$ y $P'_{4i+3}$ (ver figura 3). En los algoritmos para una sola contraseña, esto se hace con la operación

$$P_i \leftarrow P_i \oplus K_{i \mod |K|} \tag{3}$$

pero la presencia de $K^0$, $K^1$, $K^2$ y $K^3$ introduce un obstáculo: el cálculo del módulo para varias longitudes. Debido a lo engorroso que resultaría implementar este cómputo, se introdujo en pbcrypt el requerimiento de que **todas las contraseñas tengan la misma longitud en bytes**. Este valor, al igual que en Blowfish y bcrypt, se llamará $|K|$.

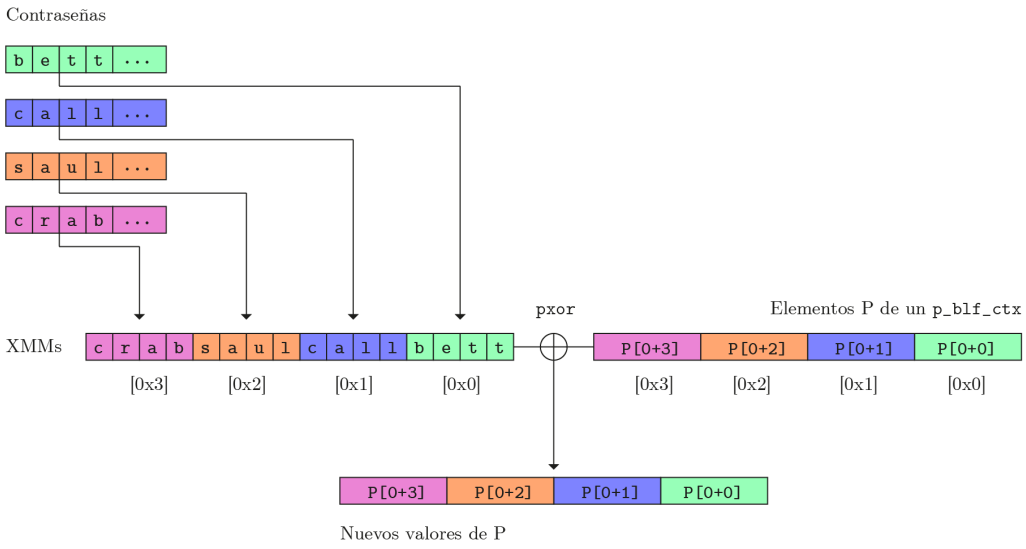


Figura 3: Cifrado XOR del arreglo $P$ en pbcrypt.

Como pbcrypt cifra varias contraseñas en simultáneo, también tiene que producir **varios *hashes***. Entonces, para que se pueda continuar usando la técnica de leer bloques contiguos de 32 bits de todas las entradas, se cambia el *hash* inicial `"OrpheanBeholderScryDoubt"` por un ***hash paralelo* de 96 bytes** en el que los bloques `"Orph"`, `"eanB"`, `"ehol"`, `"derS"`, `"cryD"` y `"oubt"` se repiten cuatro veces cada uno. Al finalizar la encriptación, este arreglo contendrá cuatro *hashes* distintos (ver figura 4).

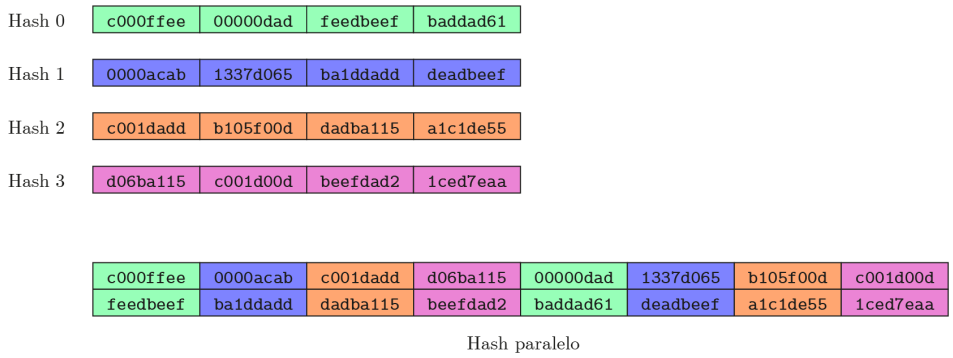


Figura 4: *Hash* paralelo.

## 3.1. Pseudocódigo

**Algorithm 3** pbcrypt

**function** PBCRYPT($K^0K^1K^2K^3, salt, C$)
    hash ← `"Orph"`*4 + `"eanB"`*4 + `"ehol"`*4 + `"derS"`*4 + `"cryD"`*4 + `"oubt"`*4
    context ← InitialParallelContext
    PBLOWFISHEXPAND(context, $K^0K^1K^2K^3, salt$)
    **for** $0 \leq i < 2^C$ **do**
        PBLOWFISH(context, $K^0K^1K^2K^3$)
        PBLOWFISH(context, $salt$)
    **for** $0 \leq i < 64$ **do**
        PBLOWFISHENCRYPT(hash, context)
    **return** hash

**function** PBLOWFISHEXPAND($context, (K^0K^1K^2K^3), salt$)
    $S' \leftarrow context.S'$
    $P' \leftarrow context.P'$
    **for** $0 \leq i < 18$ **do**
        $(P'_{4i}P'_{4i+1}P'_{4i+2}P'_{4i+3}) \leftarrow (P'_{4i}P'_{4i+1}P'_{4i+2}P'_{4i+3}) \oplus (K^0K^1K^2K^3)_{i \mod |K|}$
    $(L^0L^1L^2L^3) \leftarrow (0000)$
    $(R^0R^1R^2R^3) \leftarrow (0000)$
    **for** $0 \leq i < 18; i \leftarrow i + 2$ **do**
        $(L^0L^1L^2L^3) \leftarrow (L^0L^1L^2L^3) \oplus (salt\ salt\ salt\ salt)_{i \mod |salt|}$
        $(R^0R^1R^2R^3) \leftarrow (R^0R^1R^2R^3) \oplus (salt\ salt\ salt\ salt)_{(i+1) \mod |salt|}$
        PENCIPHER($(L^0L^1L^2L^3), (R^0R^1R^2R^3), S', P'$)
        $((P'_{4i}P'_{4i+1}P'_{4i+2}P'_{4i+3}), (P'_{4i+4}P'_{4i+5}P'_{4i+6}P'_{4i+7})) \leftarrow ((L^0L^1L^2L^3), (R^0R^1R^2R^3))$
    **for** $0 \leq i < 4$ **do**
        **for** $0 \leq j < 256; j \leftarrow j + 2$ **do**
            $(L^0L^1L^2L^3) \leftarrow (L^0L^1L^2L^3) \oplus (salt\ salt\ salt\ salt)_{i \mod |salt|}$
            $(R^0R^1R^2R^3) \leftarrow (R^0R^1R^2R^3) \oplus (salt\ salt\ salt\ salt)_{(i+1) \mod |salt|}$
            PENCIPHER($(L^0L^1L^2L^3), (R^0R^1R^2R^3), S', P'$)
            $(S'_{i,4j}S'_{i,4j+1}S'_{i,4j+2}S'_{i,4j+3}) \leftarrow (L^0L^1L^2L^3)$
            $(S'_{i,4j+4}S'_{i,4j+5}S'_{i,4j+6}S'_{i,4j+7}) \leftarrow (R^0R^1R^2R^3)$

30: **function** PBlowfish($context, K$)
31: $S' \leftarrow context.S'$
32: $P' \leftarrow context.P'$
33: **for** $0 \leq i < 18$ **do**
34: $(P'_{4i}P'_{4i+1}P'_{4i+2}P'_{4i+3}) \leftarrow (P'_{4i}P'_{4i+1}P'_{4i+2}P'_{4i+3}) \oplus (K^0K^1K^2K^3)_{i \mod |K|}$
35: $(L^0L^1L^2L^3) \leftarrow (0000)$
36: $(R^0R^1R^2R^3) \leftarrow (0000)$
37: **for** $0 \leq i < 18; i \leftarrow i + 2$ **do**
38: PEncipher$((L^0L^1L^2L^3), (R^0R^1R^2R^3), S', P')$
39: $((P'_{4i}P'_{4i+1}P'_{4i+2}P'_{4i+3}), (P'_{4i+4}P'_{4i+5}P'_{4i+6}P'_{4i+7})) \leftarrow ((L^0L^1L^2L^3), (R^0R^1R^2R^3))$
40: **for** $0 \leq i < 4$ **do**
41: **for** $0 \leq j < 256; j \leftarrow j + 2$ **do**
42: PEncipher$((L^0L^1L^2L^3), (R^0R^1R^2R^3), S', P')$
43: $(S'_{i,4j}S'_{i,4j+1}S'_{i,4j+2}S'_{i,4j+3}) \leftarrow (L^0L^1L^2L^3)$
44: $(S'_{i,4j+4}S'_{i,4j+5}S'_{i,4j+6}S'_{i,4j+7}) \leftarrow (R^0R^1R^2R^3)$

45: **function** PEncipher$((L^0L^1L^2L^3), (R^0R^1R^2R^3), S', P')$
46: **for** $0 \leq i < 16$ **do**
47: $(L^0L^1L^2L^3) \leftarrow (L^0L^1L^2L^3) \oplus (P'_{4i}P'_{4i+1}P'_{4i+2}P'_{4i+3})$
48: $(R^0R^1R^2R^3) \leftarrow \mathrm{PF}((L^0L^1L^2L^3), S') \oplus (R^0R^1R^2R^3)$
49: $((L^0L^1L^2L^3), (R^0R^1R^2R^3)) \leftarrow ((R^0R^1R^2R^3), (L^0L^1L^2L^3))$
50: $(L^0L^1L^2L^3) \leftarrow (L^0L^1L^2L^3) \oplus (P'_{64}, P'_{65}, P'_{66}, P'_{67})$
51: $(R^0R^1R^2R^3) \leftarrow (R^0R^1R^2R^3) \oplus (P'_{68}, P'_{69}, P'_{70}, P'_{71})$
52: $((L^0L^1L^2L^3), (R^0R^1R^2R^3)) \leftarrow ((R^0R^1R^2R^3), (L^0L^1L^2L^3))$

53: **function** $\mathrm{PF}((X^0X^1X^2X^3), S')$
54: $(B0^0, B1^0, B2^0, B3^0) \leftarrow (X^0_0, X^0_1, X^0_2, X^0_3)$ ▷ bytes de $X^0$
55: $(B0^1, B1^1, B2^1, B3^1) \leftarrow (X^1_0, X^1_1, X^1_2, X^1_3)$ ▷ bytes de $X^1$
56: $(B0^2, B1^2, B2^2, B3^2) \leftarrow (X^2_0, X^2_1, X^2_2, X^2_3)$ ▷ bytes de $X^2$
57: $(B0^3, B1^3, B2^3, B3^3) \leftarrow (X^3_0, X^3_1, X^3_2, X^3_3)$ ▷ bytes de $X^3$
58: **return** (
$(S'_{0,B0^0} + S'_{1,B1^0}) \oplus S'_{2,B2^0} + S'_{3,B3^0}$
$(S'_{0,B0^1} + S'_{1,B1^1}) \oplus S'_{2,B2^1} + S'_{3,B3^1}$
$(S'_{0,B0^2} + S'_{1,B1^2}) \oplus S'_{2,B2^2} + S'_{3,B3^2}$
$(S'_{0,B0^3} + S'_{1,B1^3}) \oplus S'_{2,B2^3} + S'_{3,B3^3}$
)

59: **function** PBlowfishEncrypt($hash, context$)
60: $S' \leftarrow context.S'$
61: $P' \leftarrow context.P'$
62: **for** $0 \leq i < |hash|; i \leftarrow i + 2$ **do**
63: PEncipher(
$(hash_{4i}, hash_{4i+1}, hash_{4i+2}, hash_{4i+3})$,
$(hash_{4i+4}, hash_{4i+5}, hash_{4i+6}, hash_{4i+7})$,
$S', P'$
)

## 3.2. 8 contraseñas y generalización

Tras el desarrollo de pbcrypt (subsección 4.3) y la posterior experimentación (sección 5) se diseñó una nueva variante de éste a la que se llamó **pbcrypt doble**. Dicha variante es análoga al pbcrypt explicado en esta sección, pero funciona con **ocho contraseñas en vez de cuatro**. No se profundizará mucho en su descripción, puesto que la lógica es prácticamente igual; las únicas diferencias son que todas las operaciones con cuatro *dwords* se hacen con ocho, que los registros usados para las instrucciones paralelas no miden 128 bits sino 256 y que el *hash* paralelo mide 192 bits en lugar de 96.

Lógicamente, esta extensión puede generalizarse a una **mayor cantidad de contraseñas**, con los consiguientes requerimientos para la arquitectura del procesador en el que el algoritmo vaya a correr. Para este trabajo, la máxima cantidad manejada fue 8 porque los registros más anchos de los que se disponía eran de 256 bits.

# 4. Implementación

Todo el código fuente usado para este trabajo debería estar distribuido junto con este informe; también se encuentra disponible online en https://www.github.com/cat-j/pbcrypt. Esta sección explica a grandes rasgos ciertas decisiones de diseño e implementación, pero la documentación exhaustiva de las funciones (en inglés) puede hallarse en el código.

Se desarrollaron, en principio, **tres implementaciones distintas de bcrypt** y **una implementación de pbcrypt**. Durante la experimentación, se agregaron nuevas implementaciones, incluyendo una de pbcrypt doble. A fin de evitar confusión entre estas variaciones y las versiones de bcrypt mismo (`a`, `b`, etc), se las denomina **variantes**.

## 4.1. Estructura

El proyecto consta de los siguientes directorios y archivos:

- `build/` contiene ejecutables y se genera al compilar con `make`
- `include/` contiene todos los archivos *header* (extensión `.h`)
- `scripts/` contiene scripts en Bash y Python para generar casos de test y correr experimentos
- `src/` contiene el código fuente en C y lenguaje ensamblador (extensiones `.c` y `.asm`)
  - `benchmark/` contiene código para el experimento 5.8, que no se tratará en esta sección
  - `core/` contiene las implementaciones de las funciones criptográficas estudiadas, tanto las versiones escalares como las paralelas, y una rutina que permite usarlas para cifrar datos desde la línea de comandos
  - `cracker/` contiene las rutinas principales de las tres versiones del cracker (escalar, paralelo y paralelo doble) y la funcionalidad común a todas
  - `test/` contiene las rutinas principales de los *tests*, tanto para las implementaciones escalares como las paralelas, junto con definiciones de funciones requeridas por los *tests*
  - `utils/` contiene funciones requeridas a lo largo del resto del código
- `bcrypt-macros.mac` contiene definiciones de macros usadas en el código fuente ASM
- `config-cracker` contiene opciones de configuración para los *crackers*
- `Makefile` contiene reglas para compilar los *crackers* y los *tests*

## 4.2. bcrypt: versión escalar

Lo primero en implementarse fue bcrypt con **instrucciones de propósito general de x86-64**. El código de la rutina principal `bcrypt_hashpass`, así como el de las funciones auxiliares de las que ésta se vale, se encuentra en el archivo `src/core/bcrypt.asm`. Las funciones allí definidas se traducen de forma casi directa a las exhibidas en el pseudocódigo 2:

- `blowfish_init_state_asm` copia los dígitos hexadecimales de $\pi$ a una estructura `blf_ctx` provista como argumento
- `blowfish_expand_state_asm` es una implementación de BlowfishExpand
- `blowfish_expand_0_state_asm` es una implementación de Blowfish

- `blowfish_expand_0_state_salt_asm` es una variación de la función anterior optimizada para manejar valores de $K$ de 128 bits, como es el caso de la sal
- `blowfish_encipher_register` es una implementación de ENCIPHER; también hay una versión llamada `blowfish_encipher_asm`, pero fue usada únicamente para *tests*
- `blowfish_encrypt_asm` es una implementación de BLOWFISHENCRYPT

La contraseña a hashear se representa con un *string* de C, o sea un arreglo de elementos de tipo `char` finalizado con un *null terminator*. Como el código de bcrypt de OpenBSD usa el *null terminator* para generar las subclaves, `bcrypt_hashpass` también hace esto (véase el código fuente).

El lineamiento general detrás de esta implementación de bcrypt es **minimizar los accesos a memoria**; siempre que se pueda realizar una lectura o escritura de 64 bits (en vez de 32, 16 u 8) sin complicar demasiado la implementación, esto es lo que se hace. Una excepción importante es la **lectura de la contraseña** efectuada en la macro `XOR_WITH_KEY`. Como se mencionó en el apartado 2.2, los bytes de la clave se reutilizan a medida que se generan las subclaves $P_i$. Debido a que bcrypt no impone ninguna restricción sobre la divisibilidad de la longitud en bytes de la contraseña a cifrar, puede ocurrir que un bloque de 32 bits usado para cifrar un $P_i$ consista, por ejemplo, de los dos últimos bytes de la clave seguidos del *null terminator* y el primer byte; concretamente, esta situación se da cuando el resto módulo 4 de la longitud de la contraseña es 2. Dado que estos datos **no son contiguos en memoria**, es **imposible leerlos en un solo acceso**, por lo que las claves se leen **byte por byte** y los bloques se reconstruyen usando las instrucciones `or`, `and`, `shl` y `shr`.

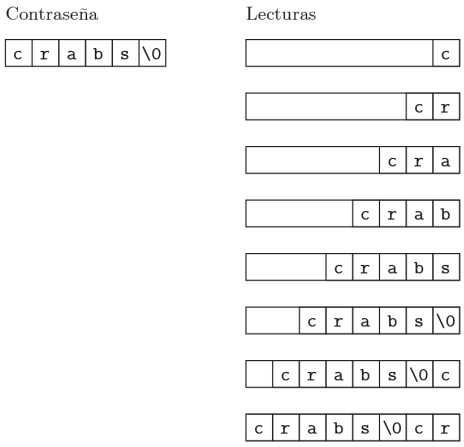


Figura 5: Lectura de una contraseña de 6 bytes (*null terminator* incluido) a un registro de 64 bits.

La técnica recién descripta toma cuatro veces más accesos a memoria que leer de a 32 bits y ocho veces más que leer de a 64, además de que requiere más instrucciones para rearmar los bloques. Por consiguiente, es de esperar que tome más tiempo de ejecución. Es debido a esto que se implementó `blowfish_expand_0_state_salt_asm`, que aprovecha la longitud en bytes divisible por 8 de la sal para cargarla en el procesador con sólo **dos accesos a memoria**.

Como copiar el contexto no implica ninguna dependencia a nivel datos, `blowfish_initstate_asm` se implementó con la instrucción de 256 bits `vmovdqa` a fin de disminuir aún más el número de accesos.

## 4.3. bcrypt: versión vectorizada

El algoritmo pbcrypt se encuentra implementado en el archivo `src/core/bcrypt-parallel.asm`. Su rutina principal se llama `bcrypt_hashpass_parallel`; el nombre pbcrypt le fue dado después de escribir esta función, por lo que se usa en este informe pero no en el código. Este archivo también cuenta con las siguientes funciones:

- `blowfish_expand_state_parallel`, `blowfish_expand_0_state_parallel`, `blowfish_expand_0_state_salt_parallel` y `blowfish_encipher_parallel` son análogas a las descriptas en el apartado 4.2, pero en vez de una sola contraseña y un `blf_ctx`, funcionan con cuatro claves y un `p_blf_ctx`
- `blowfish_parallelise_state` inicializa una estructura `p_blf_ctx` repitiendo cuatro veces cada elemento de un `blf_ctx` (ver figura 6)
- `blowfish_init_state_parallel` es análoga a `blowfish_init_state`, pero inicializa un `p_blf_ctx` a partir de un contexto paralelo inicial generado con la función anterior
- `blowfish_encrypt_parallel` genera cuatro hashes en simultáneo, con el formato descripto en el apartado 3

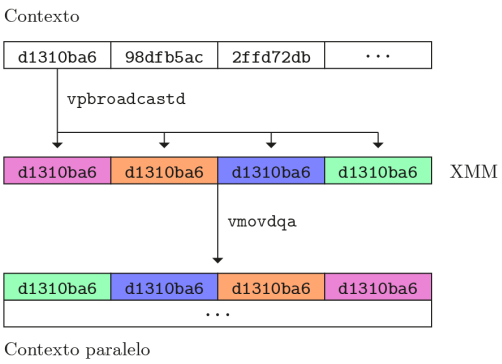


Figura 6: Uso de la instrucción `vpbroadcastd` para copiar $S_{0,0}$ a un contexto paralelo en la función `blowfish_parallelise_state`.

`blowfish_expand_state_parallel` y `blowfish_expand_0_state_parallel` contienen una implementación de la técnica de cifrado paralelo exhibida en la figura 3 de la sección 3, mientras que `blowfish_expand_0_state_salt_parallel` cuenta con una optimización similar a la descripta en la subsección 4.2, pero que usa la instrucción `vpbroadcastd` para repetir cuatro veces cada bloque de 32 bits de la sal (figura 7).

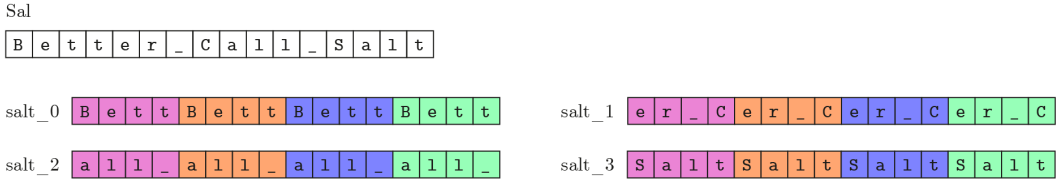


Figura 7: Paralelización de la sal en registros XMM.

Para el desarrollo de `blowfish_encipher_parallel`, resultó particularmente beneficiosa la instrucción de AVX2 `vpgatherdd` (figura 8). Dicha instrucción permite **leer memoria no contigua** usando un **registro XMM** que indique el ***offset* de cada bloque de 32 bits a cargar**; sin ella, la macro `F_XMM` sería considerablemente más larga e involucraría muchas inserciones de *dwords* en registros vectoriales. En el código de la macro, los *offsets* se obtienen usando `pslld` y `psrld` para aislar en simultáneo el mismo índice de byte de un *dword* de cada contraseña [1] y sumando a este resultado el contenido de otro registro al que se denomina `element_offset_xmm`. Las cuatro *dwords* de este último registro son, de la menos significativa a la más significativa, 0, 1, 2 y 3, ya que corresponden a los valores del índice $m$ del que se habló en la sección 3. Este esquema funciona porque los elementos $S$ y $P$ de las cuatro contraseñas se guardan de manera contigua en el contexto paralelo.

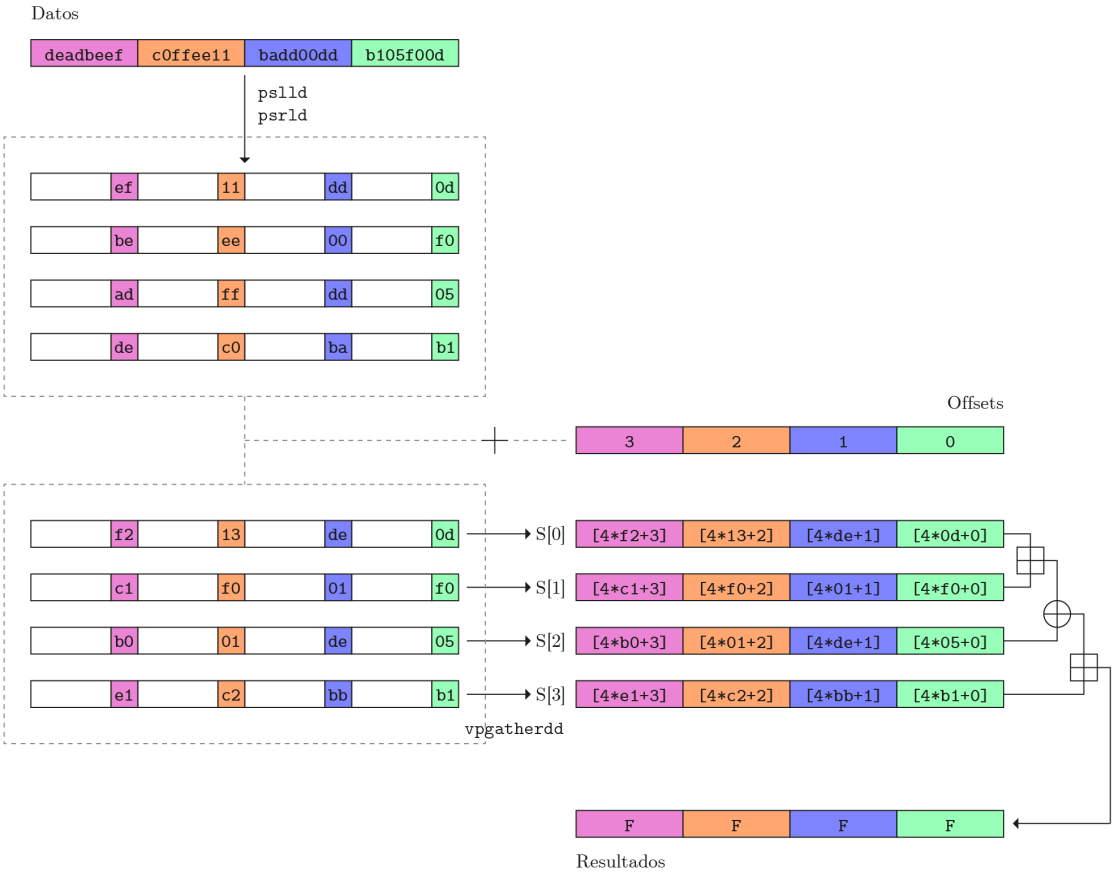


Figura 8: Cálculo vectorizado de $F$. En la implementación real, además se realiza un *shift* paralelo para multiplicar cada índice por 4 antes de sumar los *offsets*, debido a la disposición de los elementos $S$ en el contexto paralelo.

### 4.3.1. pbcrypt doble

También se implementó la variante de **pbcrypt con ocho contraseñas** descripta en la subsección 3.2; su código está en el archivo `src/core/bcrypt-parallel-double.asm`. Debido a las circunstancias de los experimentos (ver sección 5.10), pbcrypt doble fue el último algoritmo en implementarse. Nuevamente no se entra en detalles porque es análogo al pbcrypt de cuatro claves (obviamente con las modificaciones necesarias, como operar con registros YMM donde bcrypt usaría los XMM).

[1] Cada instancia de $F$ requiere un byte para acceder a algún elemento de $S$; ver algoritmo 1.

La única particularidad de la implementación de pbcrypt doble que se juzgó lo suficientemente importante para incluir en esta sección es que, en vez de `p_blf_ctx`, usa una estructura similar llamada `pd_blf_ctx`, con 2048 *dwords* en cada *S-box* y 144 en el arreglo $P$. Se consideró la posibilidad de utilizar un mismo tipo de datos para los dos algoritmos, pero las formas de hacer esto son las siguientes:

- representar $P$ y las *S-boxes* con punteros a arreglos de un número arbitrario de elementos que se define durante la ejecución (tipo de datos `uint32_t *`)
- usar algún tipo de compilación condicional para poder declarar arreglos estáticos con una cantidad paramétrica de elementos

La primera opción se descartó rápidamente, puesto que `p_blf_ctx` contiene arreglos contiguos en memoria con una cantidad fija de elementos y para cambiarla a punteros hubiera sido necesario modificar todo el código ASM ya escrito. Como esto último podría impactar en el rendimiento del programa, también se hubieran tenido que volver a correr los experimentos, la mayoría de los cuales ya habían sido ejecutados. La segunda opción no presenta este problema, pero dadas las restricciones temporales y el hecho de que en el momento no se contaba con el conocimiento técnico para implementarla, también se tuvo que dejar de lado. Sin embargo, no se descarta la posibilidad futura de realizar un *refactor* del código en el que se utilice esta última técnica.

## 4.4. Optimizaciones

### 4.4.1. *Loop unrolling*

Durante el curso de la materia, se estudió brevemente el uso de *loop unrolling* –repetir explícitamente el cuerpo de un ciclo en el código fuente, en vez de usar un contador, con el objetivo de ejecutar menos instrucciones– y su impacto, generalmente positivo, en el rendimiento de los programas escritos. Para que sea posible usar esta técnica, la **cantidad de iteraciones** del ciclo a desenrollar debe ser **conocida en tiempo de compilación o ensamblado**, ya que el programa no va a verificar condiciones en tiempo de ejecución para ver si debe volver a ciclar o no. Con la excepción de la lectura de la contraseña, **todos los ciclos de bcrypt cumplen este requerimiento**.

La primera versión de bcrypt desarrollada para este trabajo (la que se encuentra en `bcrypt.asm`) fue implementada con *loop unrolling* principalmente porque, al prescindir del contador, esta técnica requiere **menos registros**, lo cual permitió obviar algunas instrucciones `push` y `pop` y por lo tanto accesos a memoria. pbcrypt también se realizó con *loop unrolling*. En lenguaje ensamblador x86, los ciclos desenrollados tienen el aspecto

```
%rep N
<CUERPO DEL CICLO>
%endrep
```

con `N` un número natural.

Posteriormente, se programó una variante del algoritmo **sin *loop unrolling***, cuyo código fuente está en `src/core/bcrypt-no-unrolling.asm`. El propósito de esta implementación es verificar si el *loop unrolling*, además de facilitar la implementación inicial, mejora el rendimiento. Por razones que se detallarán en la subsección 4.5, todas las funciones tienen exactamente los mismos nombres que en `bcrypt.asm`, y su código es igual, con la excepción del *loop unrolling*.

### 4.4.2. Subclaves en registros YMM

A lo largo de bcrypt, las subclaves $P_0, ..., P_{17}$ son usadas varias veces, y cada vez se accede a la memoria para leerlas y sobreescribirlas. En vista de esta característica, podría resultar beneficioso **tener *P* cargado en algunos registros del procesador** y realizar todos los cálculos necesarios con ellos. Como $P$ tiene 18 elementos y cada uno mide 4 bytes, el arreglo entero ocupa **72 bytes**. Por otra parte, cada registro YMM de la extensión AVX mide **32 bytes**, por lo que alcanza con **tres registros YMM** para mantener todas las subclaves en el procesador.

Estos hallazgos motivaron el desarrollo de otra variante de bcrypt cuyo código fuente está en `src/core/bcrypt-loaded-p.asm`. Al igual que en la variante sin *loop unrolling*, las funciones que componen el algoritmo tienen el mismo nombre que en la primera implementación, pero su código tiene diferencias más importantes debido a la incorporación de $P$ en los registros YMM.

Para empezar, `bcrypt_hashpass`, tras inicializar el contexto, carga $P$ y la sal (que al medir 16 bytes entra en un único registro) a través de la macro `LOAD_SALT_AND_P`. Todas las funciones de expansión del contexto hacen uso de estos valores. El cifrado XOR de $P_0, ..., P_{15}$ –64 bytes– y la contraseña se realiza ejecutando dos veces la macro `READ_32_KEY_BYTES` y la instrucción AVX `vpxor` con el registro YMM que corresponda (primero uno que contiene $P_0, ..., P_7$ y después uno con $P_8, ..., P_{15}$). $P_{16}$ y $P_{17}$ son un caso aparte y se cifran con la macro `READ_8_KEY_BYTES` y la instrucción SSE `pxor`.

Por razones ya exploradas en la sección 2.2.1, la encriptación de cada subclave por medio de `blowfish_encipher_register` debe hacerse de a un bloque, así que en las funciones de expansión es necesario extraer repetidas veces 8 bytes de un YMM con `pextrq` e insertar las dos subclaves ya cifradas en el registro del que se extrayeron. Dado que estas instrucciones sólo operan con registros XMM y que cada XMM consiste de los 16 bytes menos significativos de un YMM, los $P_i$ se insertan con la instrucción SSE `pinsrq` en el índice correspondiente, rotando el YMM con `vpermq` para alternar entre sus dos mitades según sea necesario.

La misma `blowfish_encipher_register` también cuenta con algunas modificaciones para operar con $P$ en los YMM. Las subclaves, en vez de leerse de la memoria, se extraen con `vpextrd`, y cada ronda de Blowfish se lleva a cabo con la macro `BLOWFISH_ROUND_BIG_ENDIAN` en vez de `BLOWFISH_ROUND`. Esto último se debe a las discrepancias que presentan todas las arquitecturas de la familia x86 entre el *endianness* del procesador –y por consiguiente de los registros YMM– y el de la memoria. La figura 9 ilustra esta técnica.

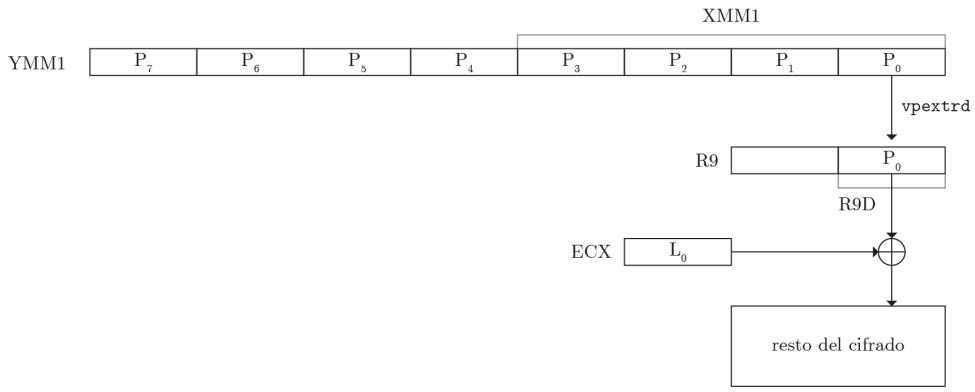


Figura 9: Uso de `vpextrd` en `blowfish_encipher_register`.

## 4.5. *Cracker*

Se escribieron dos variaciones del cracker: una basada en **bcrypt escalar**, cuya rutina principal se encuentra en `src/cracker/cracker.c`, y una basada en **pbcrypt**, que está implementada en `src/cracker/cracker-parallel.c`. Tras los experimentos con pbcrypt, se adaptó el código de `cracker-parallel.c` para otro *cracker* que hashea ocho contraseñas por vez, lógicamente llamado `cracker-parallel-double.c`; no fue posible reutilizar el código debido a la diferencia de tipos de datos mencionada más arriba, pero de realizarse el *refactor*, se usarían las mismas fuentes para ambas variantes. Las funciones usadas por los tres *crackers* están definidas en `src/cracker/cracker-common.c`.

Además de llevar a cabo su función de buscar una contraseña que resulte en un *hash* dado, los *crackers* pueden medir cuánto tiempo estuvieron hasheando textos planos hasta encontrar el deseado (o hasta que se termine la *wordlist*). Para habilitar esto, la opción `measure` del archivo `config-cracker` debe valer 1. Esta funcionalidad fue usada para los experimentos (ver sección 5).

### 4.5.1. Argumentos y precondiciones

Las dos variaciones se ejecutan desde la línea de comandos y toman como argumentos:

- un ***password record***, es decir, un *string* con un formato determinado que contiene el ***hash* de una contraseña** e información útil para la autenticación
- el nombre de un **archivo *wordlist***
- opcionalmente, un entero llamado $n_p$ (por defecto es 1024)

El *password record* se obtiene de un archivo conocido como *shadow file*; en OpenBSD se llama `master.passwd` [14]. El sistema operativo usa el *shadow file* para mantener los *hashes* de las contraseñas de los usuarios –información altamente sensible– separados de otros datos que podrían ser necesarios para agentes de menor privilegio. El archivo consiste de *records* que, en caso de pertenecer a contraseñas hasheadas con bcrypt, tienen el siguiente formato:

```
"$2V$RR$<SALT><HASH>"
```

donde el prefijo `$2V$` indica que la contraseña fue hasheada con bcrypt, `V` es la **versión específica** (`a`, `b`, `x` o `y`), `RR` es el costo representado como número en base 10, `<SALT>` es la **sal** y `<HASH>` son los **primeros 23 bytes del *hash***. Tanto la sal como el *hash* están codificados con una variación del esquema **Base64** propia de OpenBSD (puede verse el código fuente para más detalles). Así, la sal ocupa 22 bytes, el *hash* ocupa 31 y el tamaño total del *record* es **60 bytes**. A modo ilustrativo, el *record* `"$2b$08$0kTybETwGCLfZEueS0Dqb.CMzSGt65RNpTWAhxyTKzL5cVpOvT0ZE"` corresponde a la contraseña `"Go Landcrabs!"` hasheada con la versión `b` de bcrypt, con costo 8 ($2^8$ rondas de expansión) y la sal `"Better Call Salt"`.

OpenBSD usa la versión, el costo y la sal para hashear una contraseña en texto plano, comparar el resultado con el *hash* guardado en el *record* y así verificar la identidad de un usuario cuando hace falta. Los *password crackers* desarrollados en este trabajo **procesan el *record*** a través de la función `get_record_data` y se valen del **costo** y la **sal** para **cifrar todas las contraseñas** de la *wordlist* que toman.

Entre los argumentos de los *crackers* está el número $n_p$, que en el código se llama `n_passwords`. Esta variable indica la **cantidad de contraseñas en el *batch***, donde el *batch* es el *buffer* de bytes leídos por vez con `fread` a medida que se procesa la *wordlist*. Para **evitar casos borde en los *crackers* paralelos**, se introdujo la restricción de que $n_p$ sea **múltiplo de cuatro** en **pbcrypt simple** y **múltiplo de ocho** en **pbcrypt doble**. Debido a que el programa no conoce

de antemano la longitud de la *wordlist*, no es necesario que ésta sea divisible por el tamaño del *batch*, por lo que podría eventualmente tener que leer un número menor de contraseñas. En este caso, simplemente sobreescribe una porción menor del *buffer* en el que mantiene las claves y ajusta una variable para no tener que hashear de nuevo las que ya estaban en éste.

Por último, los *crackers* requieren que **todas las contraseñas de la *wordlist* sean de la misma longitud**; dicha longitud debe ser la **primera línea del archivo**. Esto se debe a la precondición de pbcrypt y pbcrypt doble sobre las cuatro u ocho claves que hashean, pero también se incluyó en el *cracker* escalar para simplificar levemente el código y no agregar un potencial *overhead* por calcular la longitud de cada una que pueda complicar las mediciones durante los experimentos (de todas formas, cuando se comparó el rendimiento de los *crackers*, se utilizaron los mismos datos de entrada para todos).

Para correr el *cracker* con los argumentos antes mencionados, deben ejecutarse los siguientes comandos en el directorio base del código fuente:

```
make cracker
./build/cracker \$2b\$08\$OkTybETwGCLfZEueSODqb.CMzSGt65RNpTWAhxyTKzL5cVpOvTOZE
↪  ./my-wordlist
```

donde `my-wordlist` obviamente debería ser un archivo que cumpla las precondiciones del *cracker*.

### 4.5.2. bcrypt

El código de `cracker.c` fue diseñado para funcionar con **cualquiera de las variantes escalares de bcrypt** desarrolladas para este trabajo. Cambiando el **código objeto linkeado** durante la **generación del ejecutable**, se puede usar la misma rutina principal para un cracker que use las funciones de `bcrypt.asm`, las de `bcrypt-no-unrolling.asm` o las de `bcrypt-loaded-p.asm`. La reglas para generar los ejecutables son `make cracker`, `make cracker-no-unrolling` y `make cracker-loaded-p`.

### 4.5.3. pbcrypt

Las únicas diferencias sustanciales entre el *cracker* paralelo y el escalar son que el paralelo llama a `bcrypt_hashpass_parallel` (pbcrypt) en vez de `bcrypt_hashpass` y que, para una misma *wordlist*, **el escalar hace cuatro veces más llamadas a función de *hash***. Para adecuarse a esta diferencia, el *cracker* paralelo tiene algunas **variaciones en la aritmética de punteros y de índices** con los que accede a las contraseñas en el *buffer*. La regla para generar su ejecutable es `make cracker-parallel`.

### 4.5.4. Aclaración sobre las versiones

Tal como se mencionó en la descripción del formato del *record*, existen cuatro versiones de bcrypt. Debido a que la única diferencia entre la versión `a` y la `b` [2] es cómo manejan el entero que representa la longitud de la clave [3] y de todas formas los *crackers* desarrollados sólo aceptan contraseñas de hasta 72 bytes –casos en los que ambas versiones deberían dar el mismo resultado–, se decidió aceptar **tanto la versión `a` como la `b`** y operar de la misma forma con la longitud en ambas.

[2] `x` e `y` se crearon para designar una implementación PHP de bcrypt que tenía un *bug* y jamás fueron soportadas por OpenBSD.

[3] A causa de un bug en la versión `a`.

## 4.6. *Tests*

De forma análoga a los *crackers*, hay **tres conjuntos de *tests* unitarios**: uno para las funciones asociadas a bcrypt escalar, uno para las asociadas a pbcrypt y otro para las asociadas a bcrypt paralelo. El código fuente de los tests de bcrypt está en `src/test/single.c` y, al igual que el *cracker*, puede usarse para testear `bcrypt.asm`, `bcrypt-no-unrolling.asm` o `bcrypt-loaded-p.asm` según se ejecute `make test`, `make test-no-unrolling` o `make test-loaded-p`. Los tests de pbcrypt están implementados en `src/test/parallel.c` y se corren con el comando `make test-parallel`. Por último, los de pbcrypt doble están en `src/test/parallel-double.c` y el comando para correrlos es `make test-parallel-double`.

Las salidas esperadas de cada caso de *test* para el código escalar se generaron con la **implementación en C de bcrypt de OpenBSD**, habiendo realizado la mínima cantidad de modificaciones necesarias para poder compilarlo y ejecutarlo junto con el código desarrollado. Puede hallarse en `src/test/openbsd.c`. Posteriormente, también se usó para comparar el rendimiento de la implementación básica en ASM con la de C. Por otro lado, las salidas esperadas de los casos de *test* del código paralelo se generaron con el código escalar una vez que éste ya había sido extensivamente testeado.

## 4.7. Cifrado con bcrypt

Para los experimentos presentados en la sección siguiente, se desarrolló un ejecutable llamado `encrypt`. Este programa simplemente usa la implementación de bcrypt contenida en `bcrypt.asm` y la codificación Base64 definida en `utils/base64.c` para **producir un *record* bcrypt** a partir de una contraseña, una sal y un costo. Estos son los comandos para ejecutar `encrypt`:

```
make encrypt
./build/encrypt "Go Landcrabs!" "Better Call Salt" 8
```

La versión de bcrypt utilizada es `b`.

# 5. Experimentos

## 5.1. Métodos y condiciones de los experimentos

Los experimentos se presentan en orden cronológico de ejecución. A menos que se indique otra cosa, el código fuente de los *crackers* y de cada variante de bcrypt se mantuvo igual entre un experimento y el siguiente. Las mediciones utilizadas y el código para procesarlas pueden hallarse en `https://github.com/cat-j/pbcrypt-jupyter`.

### 5.1.1. Especificaciones del sistema

Tanto los experimentos como la generación de datos de entrada se llevaron a cabo en una máquina con un procesador **Intel Core i5-7600 de 64 bits** y **16 GB de RAM** corriendo **Manjaro Linux** con ***kernel* 4.9.183**. Se usó **NASM 2.14.02** para ensamblar y **GCC 9.1.0** para compilar. Las especificaciones completas pueden verse en el apéndice A.

### 5.1.2. Generación de datos de entrada

Para generar las *wordlists* usadas en los experimentos, primero hace falta descargar el archivo `realhuman_phill.txt` [4] de este link y extraerlo a una carpeta llamada `wordlists/` en el directorio base del código fuente. Esta lista fue elegida porque su descomunal tamaño (683 MB) permite producir *wordlists* de características muy variables en lo que respecta a su longitud y la de las contraseñas que las forman; además, dado que contiene datos presuntamente tomados de **usuarios reales**, se considera que es similar a las *wordlists* que podrían usarse en un **contexto real** de **seguridad informática ofensiva**.

Una vez descargado el archivo, los datos de entrada para los experimentos pueden generarse a través de los comandos

```
python3 ./scripts/split-wordlist.py ./wordlists/realhuman-phill.txt
./scripts/generate-test-cases.sh
```

El primero separa `realhuman_phill.txt` en varias *wordlists* que cumplen con las **precondiciones de los *crackers***: cada una tiene contraseñas de una única longitud en bytes y su primera línea es esta longitud. El segundo usa estas *wordlists* para producir datos de entrada en los que la **longitud de la contraseña** y el **tamaño de la lista** tengan los valores **requeridos por cada experimento** [5].

La última contraseña de cada *wordlist* es siempre alguna variación de `"Go Landcrabs!"` con el número requerido de bytes, creada a partir de la función de Python `generate_password`. La motivación detrás de esto último es que vuelve relativamente sencillo automatizar experimentos en los que los *crackers* siempre tengan éxito, pues la misma contraseña se usa para producir un *record* (con la sal `"Better Call Salt"`) y crackearlo con la *wordlist* correspondiente. También podrían realizarse experimentos para medir el rendimiento puramente en términos de contraseñas hasheadas sin necesidad de que se encuentre la que se busca, pero también se quiso incorporar la influencia de la comparación de *hashes* y se buscó que alguno coincidiera con el del *record*. Todos los *records* se generaron con el binario `encrypt` mencionado en la subsección 4.7.

[4] Originalmente obtenido de `https://chrisreeves.co.nz/2014/06/16/collection-wordlists/`, aunque no pudo hallarse ninguna mención de las fuentes de las que se relevaron las contraseñas en sí.

[5] Puede ser que dos experimentos distintos requieran el mismo par de valores, pero como los *scripts* son determinísticos, la *wordlist* generada es siempre igual y se usa la misma para los dos experimentos.

### 5.1.3. Ejecución de experimentos

Los *scripts* para correr todos estos experimentos se encuentran en `scripts/run-experiments.sh`. Cada uno de ellos se ejecutó en una terminal de Linux sin tener corriendo ningún otro proceso, con la excepción de *daemons* necesarios para mantener el sistema en funcionamiento. A pesar de que `run-experiments.sh` contiene el código de todos los experimentos, en la práctica no se llevaron todos a cabo con una única ejecución del *script*, sino que se comentaron y descomentaron las líneas para correr cada uno según fuera necesario. Esto se debe principalmente a que los hallazgos de ciertos experimentos motivaron el diseño de algunos de los posteriores.

## 5.2. Complejidad temporal, *wordlist* y contraseña

El objetivo de este experimento es tener un panorama general del rendimiento de cada variante del *password cracker* en función de la **longitud de la *wordlist***, así como de la influencia de la longitud de la contraseña. La función para ejecutarlo se llama `experiment_growing_wordlist`.

Se llevó a cabo con **contraseñas de tres longitudes distintas**: **72, 13 y 3** (en orden de realización). El **costo** se fijó en **8**, porque se consideró un número lo suficientemente alto para que hashear una clave tuviera cierto costo computacional, pero lo suficientemente bajo para que el tiempo de ejecución del experimento fuera práctico. Los tamaños de las *wordlists* usadas van **desde 32 hasta 8192 contraseñas, en saltos de 32** y el valor de $n_p$ es 16. Se midió únicamente el rendimiento de los cuatro *crackers* iniciales descriptos en la sección 4; las demás variantes no fueron desarrolladas hasta después de la realización de este experimento y de los tres siguientes.

Cada uno de los *password crackers* procesó una única vez cada *wordlist*; esta decisión de diseño se debe principalmente a que el experimento tardó varios días en ejecutar y se juzgó que el gran número de contraseñas hasheadas alcanza para considerar que se tienen muestras significativas. Dada la consistencia de los resultados, tampoco hizo falta repetir la ejecución.

### 5.2.1. Tamaño de la *wordlist*

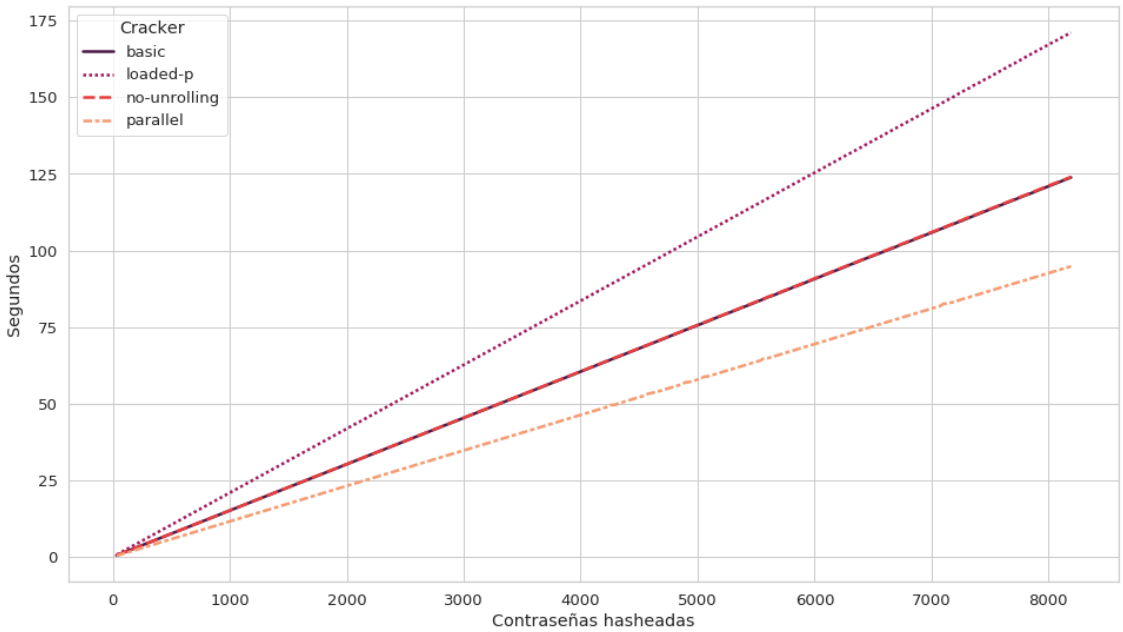


Figura 10: Rendimiento de los *crackers* para *wordlists* de longitudes 32 a 8192 y contraseñas de longitud 13. Costo 8.

La figura 10 muestra que para todos los *crackers* el tiempo de ejecución crece de manera **lineal** en función del tamaño de la *wordlist*, lo cual no es sorprendente ya que cada contraseña simplemente le **suma** un tiempo de procesamiento que no tendría por qué variar para una misma longitud. Resulta mucho más interesante analizar las **diferencias** –o falta de ellas– en el **rendimiento de cada *cracker***.

**Vectorización.** Para empezar, puede considerarse a pbcrypt **un éxito relativo** con respecto a la implementación escalar en ASM: mientras que la versión básica tarda **123.833729 segundos** en procesar la *wordlist* de 8192 contraseñas, la paralela tarda **94.813735 segundos**. Esto implica una **disminución del 23.434 % en el tiempo de ejecución**. Idealmente, dado que hashea cuatro claves a la vez, pbcrypt sería cuatro veces más rápido (disminución del 75 %), pero evidentemente hay cosas de esta implementación particular que agregan **algún tipo de *overhead*** por el que la diferencia no es tan alta. En algunos experimentos posteriores se estudiaron algunas de las posibles formas que podría tener este *overhead*.

***Loop unrolling.*** La segunda observación es que el **rendimiento del *cracker* con *loop unrolling*** es **casi igual** al del *cracker* sin *loop unrolling*. El primero es consistentemente más rápido, pero la diferencia no suele superar el décimo de segundo. Algunas hipótesis –que no tienen por qué ser mutuamente excluyentes– de por qué la mejora es tan poco significativa son:

- El **cuello de botella** más importante de bcrypt **no es** el *loop overhead* (las sumas/restas y comparaciones usadas para el loop).
- El **número de iteraciones** de los ciclos desenrollados **no es lo suficientemente grande** para que la técnica resulte en una mejora considerable (esta hipótesis está íntimamente ligada con la anterior).
- El **mayor tamaño del binario con *unrolling*** [6] resulta en una **proporción más alta de *cache misses*** y por consiguiente agrega otro *overhead*.
- Las **dependencias a nivel datos** impiden el uso de *pipelining* en el procesador, por lo cual el *unrolling* no elimina el *overhead* asociado con la presencia de saltos condicionales en los ciclos [7].

La sección sobre *loop unrolling* de la guía de optimización de rutinas en ASM de Agner Fog [3] plantea algunos de estos factores. Dadas las condiciones particulares de estas implementaciones, **la primera y la segunda resultan plausibles**: la mayoría de los ciclos tienen **números muy bajos de iteraciones**, frecuentemente 9 o menos (pues consisten en hacer operaciones sobre el arreglo $P$). El más largo es de 512 iteraciones y es el único ciclo de esta longitud.

**Arreglo P en registros YMM.** Contrariamente a lo deseado, el tiempo de ejecución de `cracker-loaded-p` es bastante más alto que el del *cracker* básico; tarda aproximadamente 75 % más en procesar una *wordlist* del mismo tamaño. Dada la suposición de que iba a ser más eficiente por ahorrarse accesos a memoria, este resultado es una sorpresa. Se plantean dos hipótesis para explicarlo:

- Hay algún tipo de **penalización** por usar instrucciones **AVX y SSE** en proximidad.
- Las instrucciones que se usan para manejar la sal y las subclaves cargadas en los registros YMM y XMM **no son más rápidas que los accesos a memoria**.

Ambas fueron analizadas en experimentos posteriores (subsecciones 5.6 y 5.8 respectivamente), por lo que no se desarrollarán en profundidad en esta sección.

[6] `cracker` pesa 88KB, mientras que `cracker-no-unrolling` pesa 60KB.
[7] Sin tener en cuenta el uso de *branch prediction*.

### 5.2.2. Longitud de la contraseña

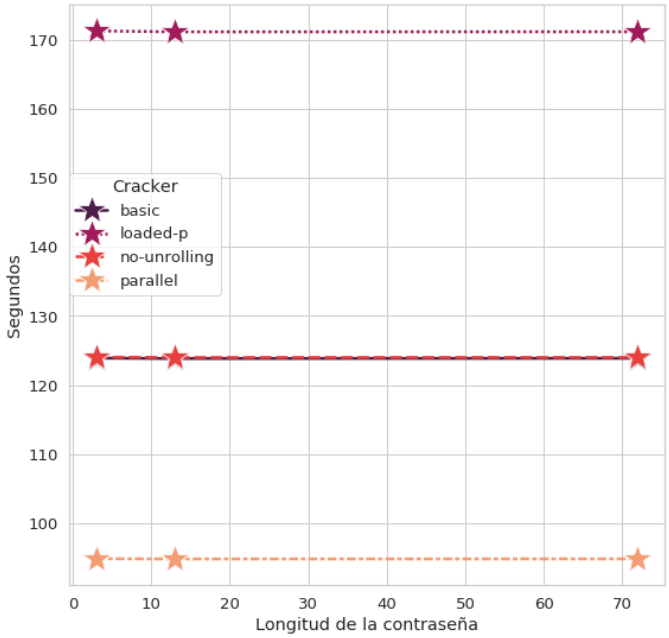


Figura 11: Rendimiento de los *crackers* para contraseñas de longitudes 3, 13 y 72 y una *wordlist* de longitud 8192.

Como puede verse en la figura 11, la longitud de la contraseña **no influye** en el rendimiento de los *crackers*. Esto indica que la técnica de lectura de contraseñas de la figura 5 (subsección 4.2) **no es particularmente costosa**, por lo menos en relación a otras subrutinas de bcrypt y pbcrypt. El mismo análisis se realizó para las *wordlists* de longitudes 32, 64, 128 y 256 y los resultados fueron **muy similares**, por lo que no se incluyen los gráficos correspondientes en este apartado.

Considerando que los ciclos de lectura de contraseñas no se desenrollaron en ninguna variante, estos resultados constituyen **evidencia a favor** de la hipótesis de que **el *loop overhead* no es significativo en bcrypt.** Si lo fuera, ejecutar 8192 veces un ciclo de 72 iteraciones –un número 24 veces más grande que 3– tendría un impacto perceptible en el tiempo de procesamiento, al contrario que lo indicado por estas observaciones.

## 5.3. Complejidad temporal, rondas

Este experimento pretende analizar el impacto de la **paralelización** en el arma más potente de bcrypt: **el costo** (parámetro $C$). Por supuesto que no es esperable que disminuir el tiempo de ejecución en un **factor constante** –como ocurre en el *cracker* paralelo con respecto al escalar– introduzca un cambio cualitativo en el rendimiento cuando entra en juego algo de **complejidad exponencial**, en este caso el ***key setup* en función de las rondas**; sin embargo, en la práctica, la optimización podría llegar a tener efectos **no despreciables** en el rendimiento. La función para ejecutar el experimento es `experiment_rounds`.

Esta vez, se usó una única ***wordlist* de 1024 contraseñas de 13 bytes** cada una, y cada *password cracker* la procesó para crackear contraseñas hasheadas con un **costo de entre 4 y 16**, i.e. $2^4, 2^5, ..., 2^{16}$ rondas de *key setup*, siempre con un *batch* de 16 claves. Al igual que ocurrió con la longitud de la *wordlist* y la clave en el primer experimento, cada binario se corrió una sola vez por cantidad de rondas, debido a que de otra forma el tiempo total de ejecución no habría sido manejable.

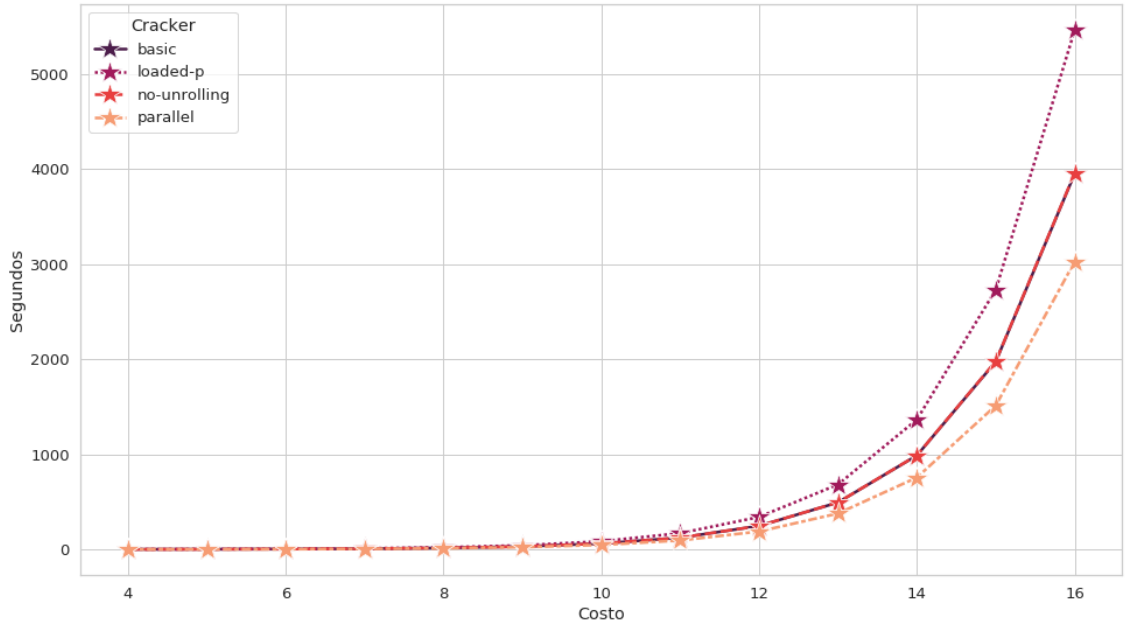


Figura 12: Rendimiento de los *crackers* para costos de 4 a 16. *Wordlist* de 1024 contraseñas.

Los resultados exhibidos en la figura 12 son consistentes con lo supuesto al principio: por más que el *cracker* con pbcrypt es considerablemente más rápido que el básico, **no mitiga la complejidad exponencial** del *key setup*. Esto evidencia la **efectividad** del **parámetro *C*** como protección contra los **ataques de fuerza bruta**.

De todas formas, pbcrypt terminó siendo bastante beneficioso: tardó aproximadamente 15 minutos menos en crackear una contraseña encriptada con costo 16. En el contexto de un *password cracker* de uso industrial con una *wordlist* mucho más grande, podría ahorrarse bastante tiempo de trabajo (sin considerar el *overhead* de procesar los datos de entrada para cumplir con los requerimientos de pbcrypt).

## 5.4. Impacto del tamaño del *batch* en el rendimiento

**Aclaraciones preliminares.** Durante la realización de este experimento, se encontró un *bug* en la rutina principal del *cracker* paralelo. Cuando leen el último *batch* de contraseñas, los *crackers* ajustan dos variables enteras llamadas `batch_size` y `n_passwords`; el vectorizado además ajusta otra llamada `password_groups`. Esto se hace para manejar casos borde en los que la cantidad de contraseñas de la *wordlist* no es divisible por $n_p$, y por lo tanto el *cracker* debe procesar un número menor de claves en la última lectura que hace al archivo. El *bug* consistía en un *off-by-one error* (error por un paso) debido al cual, si $n_p$ no dividía a la longitud de la *wordlist*, no hasheaba las últimas cuatro claves. Como este *bug* **no afectaba las ejecuciones de los experimentos previos**, se decidió **mantener los resultados** ya obtenidos en vez de repetirlos; a partir de este experimento, se usó siempre el código de los *crackers* con el *bug* corregido.

**Experimento.** Tal como se indica en el nombre de la subsección, el objetivo es simplemente medir el efecto de variar el tamaño del *batch* en el tiempo de ejecución de los *password crackers*. Esta vez se crackeó una contraseña cifrada con **costo 8** usando una ***wordlist* de 8192 contraseñas de 13 bytes** y $n_p$ fue desde 64 hasta 8192 contraseñas en saltos de 64. Sin embargo, durante el procesamiento de los datos, se observó que el valor escrito en las mediciones difería del $n_p$ efectivamente usado, debido a otro *bug* que no afecta la correctitud de los programas pero sí la calidad de los datos obtenidos; concretamente, cuando la longitud de la *wordlist* no es divisible por

$n_p$, el valor de `n_passwords` que se escribe es el ajustado en vez del original. Por lo tanto, para el análisis del experimento se descartaron las mediciones en las que $n_p$ no dividiera a 8192, y se usaron sólo las correspondientes a **64, 128, 512, 1024, 2048, 4096 y 8192**. El experimento se ejecuta con `experiment_growing_batch`.

Mientras que en los experimentos anteriores se analizó únicamente el tiempo que los *crackers* pasaron hasheando contraseñas, en este se estudió el **tiempo total de ejecución** desde que abrieron la *wordlist* hasta que terminaron. Esto se hizo porque no es razonable suponer que la cantidad total de contraseñas cargadas en un *buffer* afecte el tiempo que se tarda en hashear una sola (o cuatro), pero sí cabe preguntarse si a nivel del *cracker* como programa existe algún *trade-off* entre hacer más llamadas a `fread` y manejar un *buffer* más grande.

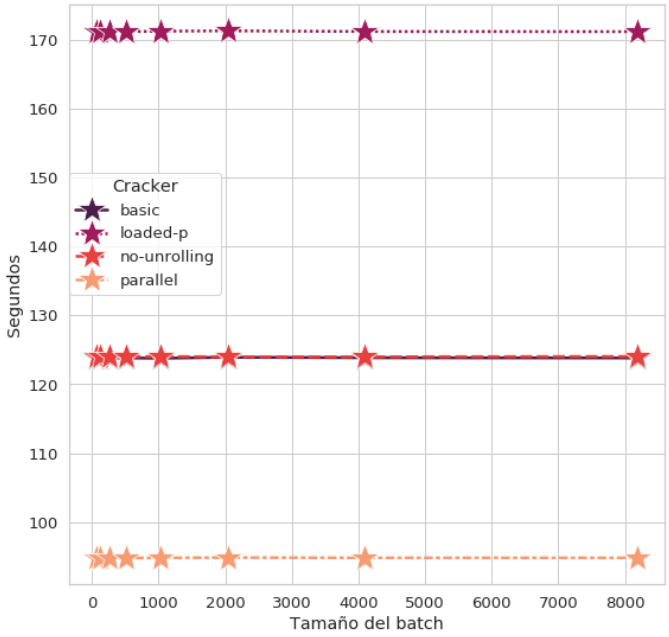


Figura 13: Rendimiento de los *crackers* para *batches* de tamaños 64, 128, 512, 1024, 2048, 4096 y 8192, contraseña de longitud 13 y *wordlist* de longitud 8192. Costo 8. Tiempo total de ejecución (no sólo *hashing*).

El gráfico de la figura 13 evidencia que **el tamaño del *batch* no tiene un impacto perceptible en el rendimiento**, por lo menos para el rango de valores aquí manejado. Sería interesante para trabajos futuros repetir esta experiencia con valores de $n_p$ mucho mayores, pero en este caso, el tiempo de procesamiento de una *wordlist* más grande resultó prohibitivo.

## 5.5. Alineamiento de código a 64 bytes

En el análisis del primer experimento realizado (subsección 5.2) se ve que el impacto del *loop unrolling* en el rendimiento del *cracker* es ínfimo. Entre las posibles causas planteadas está la potencial **proporción de *cache misses*** debida al mayor tamaño del binario con *unrolling*. A fin de verificar si la memoria *cache* efectivamente influye en el tiempo de ejecución de cada *cracker*, se agregaron **cuatro variantes nuevas**: `cracker-aligned`, `cracker-no-unrolling-aligned`, `cracker-loaded-p-aligned` y `cracker-parallel-aligned`.

Fog [4] trata el impacto de las fallas de *cache* en el rendimiento de los programas y algunas técnicas para prevenirlas. Una de ellas es la **alineación del código**: como el procesador busca el código de la memoria en bloques, puede ser beneficioso que esté alineado en ellos de manera que se ahorre algunos accesos a memoria. En este caso, como la memoria *cache* del procesador utilizado tiene

líneas de 64 bytes (512 bits), se usó la directiva `align 512` antes de la definición de cada función en los archivos `.asm`. Se reutilizó el código de los *crackers* desarrollados inicialmente, con la excepción del agregado de esta directiva y el linkeo de las definiciones correspondientes.

Las *wordlists* usadas para el experimento tienen desde **128 hasta 8192 contraseñas de 13 bytes, en saltos de 128**. Nuevamente la contraseña se cifró con **costo 8** y se crackeó con 16 claves por *batch*. La función para correr el experimento es `experiment_growing_wordlist_aligned`.

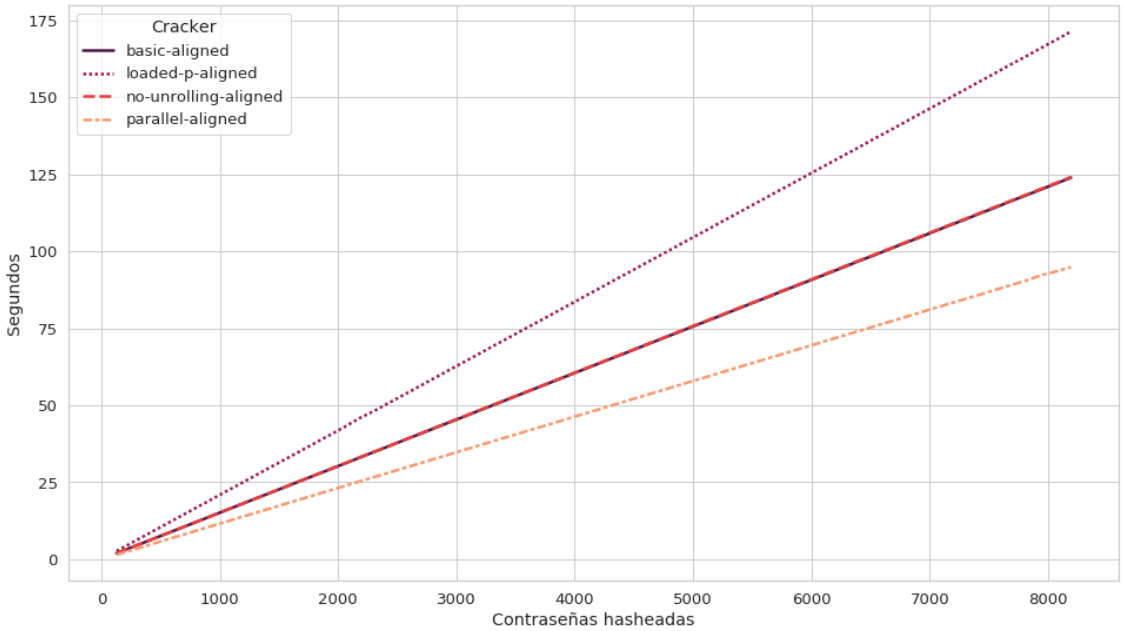


Figura 14: Rendimiento de los *crackers* con código alineado a 64 bytes para *wordlists* de longitudes 128 a 8192. Costo 8.

En la figura 14 puede verse que alinear el código a 64 bytes **no tuvo ningún impacto** en la diferencia entre el rendimiento del *cracker* con *loop unrolling* y el que no usa *unrolling*. De hecho, tampoco tuvo **ningún efecto visible sobre el rendimiento general**, ya que los tiempos de procesamiento son prácticamente iguales a los de la figura 10. Considerando que el trabajo citado aclara que **el impacto del alineamiento del código es mínimo en la mayoría de los casos**, estos resultados no son sorprendentes. Tampoco se puede descartar que **siga habiendo *cache misses*** debido a código o datos no alineados en **otra parte de los programas**, puesto que no se modificó ningún otro aspecto del código fuente; además, las estructuras `blf_ctx` y `p_blf_ctx` ya siguen las reglas de alineamiento propuestas por Fog [4]. Otra opción posible es que **el código ya estuviera alineado desde antes**: los tamaños de los binarios son **iguales a los no alineados**, con la excepción de `cracker-parallel-aligned` (que pesa 116KB en vez de 112KB; de todas formas no es una diferencia significativa).

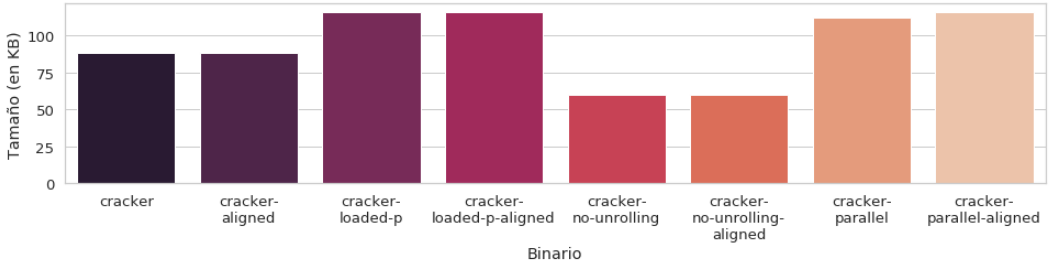


Figura 15: Tamaño de los ejecutables.

## 5.6. Penalización por transiciones AVX-SSE

En la subsección 5.2 también se trata el alto tiempo de procesamiento de `cracker-loaded-p`, y una de las hipótesis que se plantean para explicarlo es que entra en juego la penalización por **intercalar instrucciones** pertenecientes a la extensión **AVX** –las que tienen el prefijo `VP`– con otras que pertenecen a **SSE** –prefijo `P`–. Efectivamente, el código de `bcrypt-loaded-p.asm` usa ambos tipos de instrucciones en proximidad, como puede verse en los siguientes fragmentos del desensamblado de `bcrypt-loaded-p.o`:

```
0000000000001a88 <blowfish_expand_state_asm.p_array_salt>:
    [...]
    1abf: c4 e3 fd 00 c9 4e     vpermq $0x4e,%ymm1,%ymm1
    1ac5: 66 49 0f 3a 22 cd 00  pinsrq $0x0,%r13,%xmm1
    1acc: c4 e3 fd 00 c9 4e     vpermq $0x4e,%ymm1,%ymm1

000000000000aaaa <blowfish_expand_0_state_salt_asm.p_array_data>:
    [...]
    aaca: c4 e3 fd 00 c9 4e     vpermq $0x4e,%ymm1,%ymm1
    aad0: 66 49 0f 3a 22 cd 00  pinsrq $0x0,%r13,%xmm1
    aad7: c4 e3 fd 00 c9 4e     vpermq $0x4e,%ymm1,%ymm1

000000000000ef28 <blowfish_encrypt_asm.do_encrypt>:
    [...]
    ef47: 66 49 0f 3a 22 e5 01  pinsrq $0x1,%r13,%xmm4
    ef4e: c4 e3 fd 00 e4 4e     vpermq $0x4e,%ymm4,%ymm4
    ef54: 66 49 0f 3a 16 e5 00  pextrq $0x0,%xmm4,%r13
    ef5b: e8 29 18 ff ff        callq  789 <blowfish_encipher_register>
    ef60: 66 49 0f 3a 22 e5 00  pinsrq $0x0,%r13,%xmm4
    ef67: c4 e3 fd 00 e4 4e     vpermq $0x4e,%ymm4,%ymm4
```

En particular, `blowfish_expand_state_asm` y `blowfish_expand_0_state_salt_asm` repiten la secuencia `vpermq`, `pinsrq` muchas veces. Además, lo hacen con el **mismo registro** alternando entre **su versión YMM y su versión XMM**, por lo cual el procesador necesita preservar los 128 bits más altos. Este es el tipo de situación en el que **puede haber overhead asociado a las penalizaciones AVX-SSE**.

A fin de combatir estas penalizaciones, se desarrolló una nueva variante de bcrypt que se encuentra en el archivo `bcrypt-loaded-p-no-penalties.asm`, y también se creó una regla en `Makefile` para su *cracker* asociado `cracker-loaded-p-no-penalties`. Esta implementación tiene las siguientes diferencias fundamentales con respecto a `bcrypt-loaded-p.asm`:

- **todas las instrucciones SSE** se reemplazaron por **sus equivalentes AVX**
- para evitar tener que preservar los bits más altos de los registros YMM, **las subclaves se cargan en 5 registros XMM**

Se comparó el rendimiento de `cracker-loaded-p-no-penalties` con el de `cracker-loaded-p` y `cracker` usando wordlists de **entre 32 y 8192 contraseñas de 13 bytes, con saltos de 32**. El **costo** usado para la encriptación es **8** y $n_p$ es 16. El experimento se corre con la función `experiment_no_penalties`.

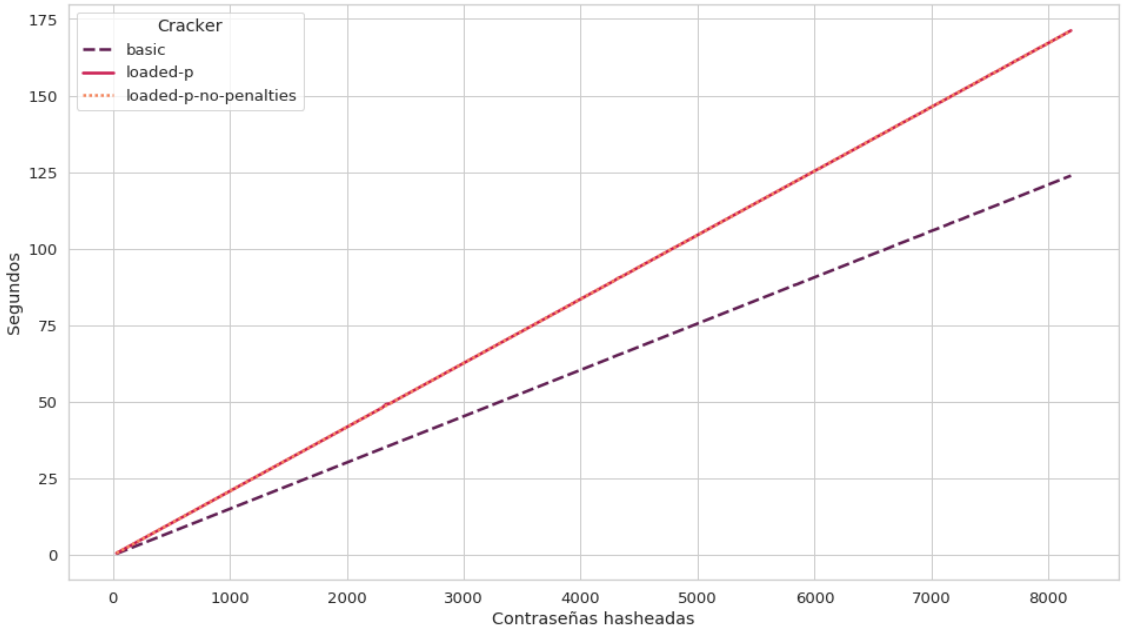


Figura 16: Rendimiento de los *crackers* para *wordlists* de longitudes 32 a 8192 y contraseñas de 13 bytes. Costo 8.

La figura 16 indica que las modificaciones realizadas **no tienen un impacto observable en el tiempo de procesamiento**, lo cual constituye **evidencia en contra de la hipótesis** de que el bajo rendimiento se debía a penalizaciones por transiciones AVX-SSE.

## 5.7. Comparación con la implementación de OpenBSD

En ciertos casos, el rendimiento de los programas escritos directamente en lenguaje ensamblador es **superior** al de sus equivalentes compilados a partir de código en C. Esto puede deberse a factores como el *overhead* que se produce cuando el compilador introduce más instrucciones para, por ejemplo, implementar la guarda de un ciclo. El compilador GCC brinda la posibilidad de **optimizar** el código generado para deshacerse de estos problemas y potencialmente superar el rendimiento del programa ASM a través de las opciones `-O0`, `-O1`, `-O2` y `-O3`. Al igual que las mejoras unidas a la vectorización, este fenómeno fue estudiado durante la cursada de Organización del Computador II.

Este experimento se desarrolló para comparar el rendimiento de dos de las implementaciones ASM desarrolladas inicialmente y un **código C compilado con niveles de optimización desde 0** (mayor tiempo esperado de ejecución) hasta **3** (menor tiempo esperado de ejecución). La función para ejecutarlo es `experiment_openbsd`.

Las variantes estudiadas son únicamente **bcrypt básico** y **pbcrypt**, ya que como los experimentos anteriores revelaron que el *loop unrolling* es conveniente [8] y el rendimiento de `cracker-loaded-p` no es bueno, se dejó de trabajar con este último ejecutable y con `no-unrolling`. Como ya se contaba con la implementación de bcrypt en C de OpenBSD usada para los *tests*, se usó este mismo código. El esqueleto del *password cracker* es el mismo `cracker.c` ya escrito, pero durante su compilación se linkea el código C. Además, puede compilarse con cualquiera de los niveles de optimización que GCC permite, por lo que hay cuatro reglas distintas (todas con el formato `make cracker-openbsd-ON`, donde `N` es un número entre 0 y 3).

[8] Más que nada por el hecho de que permite usar menos registros y así simplificar ciertos aspectos de la escritura del código. El rendimiento, como evidenciaron los experimentos anteriores, no se ve afectado.

Los datos de entrada del experimento consisten en **wordlists de longitud 32 a 2048** y una **contraseña de 13 bytes** cifrada con **costo 8**. El $n_p$ usado fue nuevamente 16.

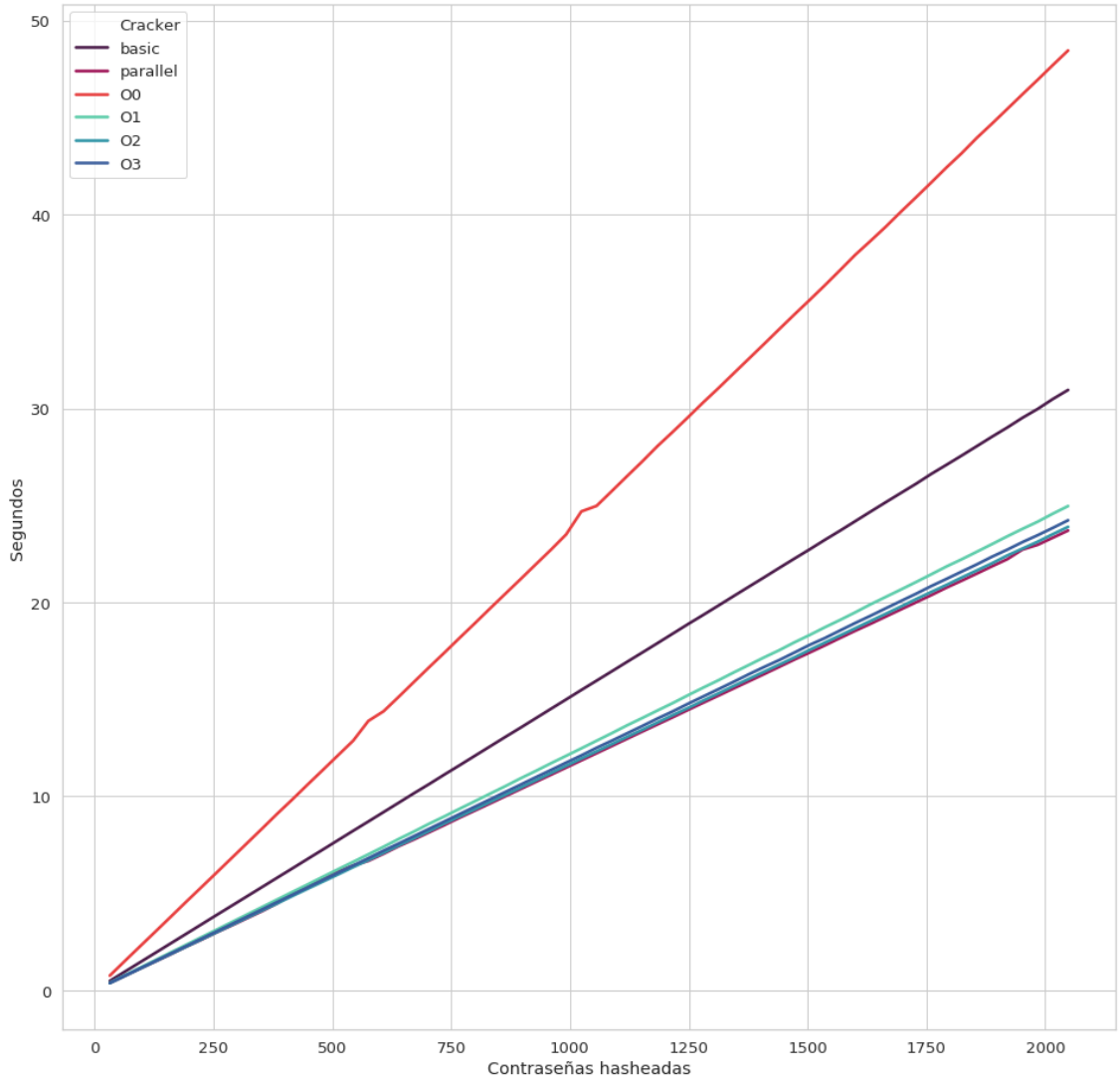


Figura 17: Rendimiento de los *crackers* para *wordlists* de longitudes 32 a 2048 y contraseñas de 13 bytes. Costo 8.

En la figura 17 puede observarse que la *performance* del bcrypt **básico** desarrollado para el trabajo **supera ampliamente** la del código compilado **sin optimizaciones**, pero con sólo subir el nivel de optimización de 0 a 1, **el código compilado ya es mucho más veloz**. Los niveles 2 y 3 no introducen mejoras considerables; de hecho, el rendimiento del código compilado con `-O2` es levemente superior al del compilado con `-O3`. Para comprender en profundidad los motivos de estos últimos dos fenómenos se requiere potencialmente un análisis fino del código desensamblado, lo cual se escapa del alcance de este trabajo.

Otra observación que se desprende de este experimento es que, para las *wordlists* de mayor tamaño, el rendimiento del *cracker* pbcrypt es **sólo levemente superior** al del *cracker* C con nivel 2 de optimización, mientras que para *wordlists* no tan grandes ni siquiera puede percibirse una diferencia. Se buscaron instrucciones SIMD en el código desensamblado de `cracker-openbsd-O2` para ver si esto podía influir en que su rendimiento sea comparable al de un programa que procesa cuatro veces más datos en gran parte de sus operaciones, pero no se halló ninguna.

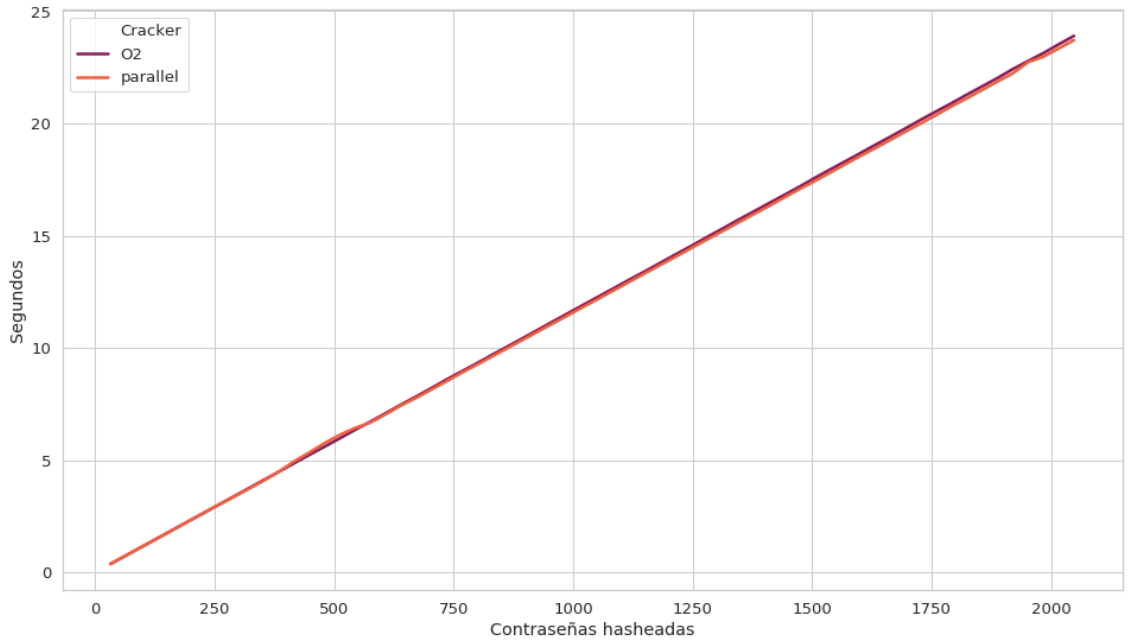


Figura 18: Rendimiento de los *crackers* para *wordlists* de longitudes 32 a 2048 y contraseñas de 13 bytes. Costo 8.

La figura 18 muestra un detalle de las mediciones del *cracker* pbcrypt y el *cracker* C más eficiente. La diferencia en el rendimiento es ínfima; de hecho, en unas pocas *wordlists* con alrededor de 500 contraseñas, el código compilado tuvo mejor *performance*.

## 5.8. Comparación entre instrucciones SIMD y accesos a memoria

Otra posible causa del bajo rendimiento de `cracker-loaded-p` aludida en la subsección 5.2 es que las instrucciones utilizadas para acceder a las subclaves en los registros YMM **no son realmente más veloces que los accesos a memoria**. Este experimento tiene el objetivo de **medir el tiempo promedio de ejecución de cada instrucción involucrada** para verificar si este realmente es el caso.

El código fuente usado para ejecutar las instrucciones se encuentra en la carpeta `benchmark/`. El archivo `instructions.asm` contiene funciones para ejecutar una cantidad de veces dada las instrucciones `vpextrd`, `pextrq`, `vpextrq`, `pinsrq`, `vpinsrq`, `vpermq`, `vpshufb`, `bswap`, `mov` y `vmovdqa` de 128 bits, estas últimas tanto para lectura como para escritura. `bswap` no se usa para manejar los registros YMM, pero sí para la macro `REVERSE_ENDIANNESS_2_DWORDS_BSWAP`, que se usa exclusivamente en las variantes de bcrypt con las subclaves cargadas en ellos. `benchmark.c` consiste de funciones para medir el tiempo de ejecución de las de `instructions.asm` y una rutina principal para correr estos procedimientos. Tanto las lecturas como las escrituras se realizaron con registros de propósito general de 64 bits. El experimento se corre simplemente ejecutando `./build/benchmark <ARCHIVO DE MEDICIONES>`.

Para medir el tiempo de ejecución promedio, cada instrucción se repitió $2^{32}$ veces. En el sistema operativo usado, la función de C `clock()` devuelve microsegundos, así que esta es la unidad usada para las mediciones; dado que el foco del experimento es medir la **diferencia** entre el tiempo de ejecución de cada instrucción, no se considera que la unidad en sí sea tan importante.

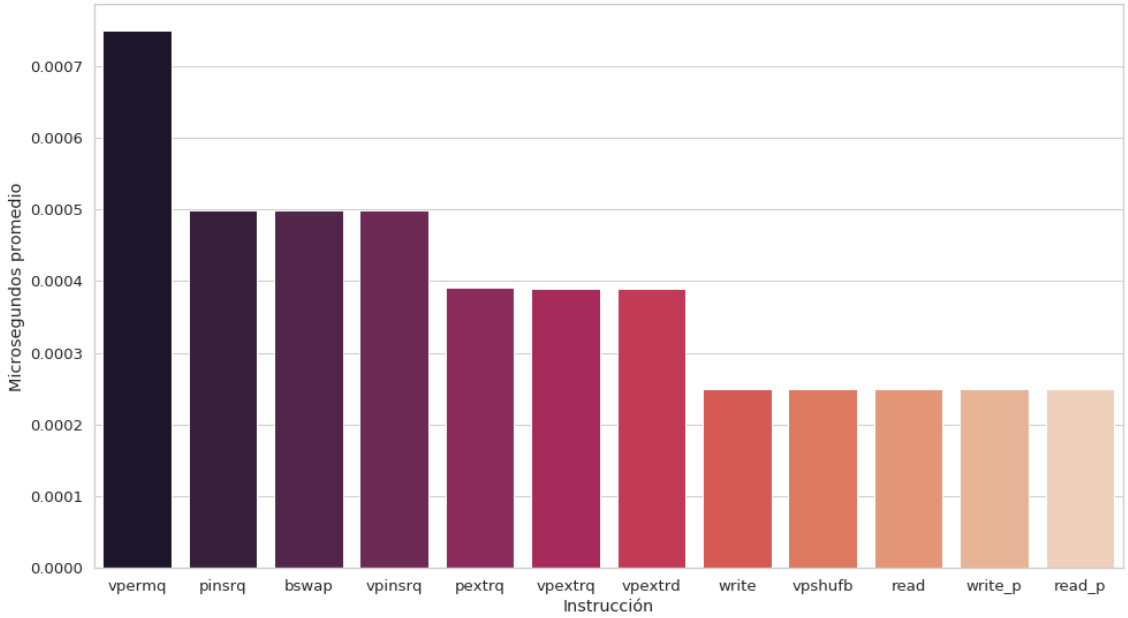


Figura 19: Microsegundos promedio por instrucción (cada una se ejecutó $2^{32}$ veces).

Los resultados graficados en la figura 19 muestran que `vpermq` [9], que se usa en las variantes con subclaves cargadas para **permutar las *quadwords*** [10] de los registros YMM y así acceder a su parte superior, es por lejos **la instrucción más costosa** en términos de tiempo. **Insertar 64 bits** con `pinsrq` o `vpinsrq` tarda aproximadamente **el doble que escribirlos a la memoria** con `mov`; la diferencia entre **extraerlos** con `pextrq` o `vpextrq` y **leerlos de la memoria** no es tan amplia, pero de todas formas es muy significativa y también **favorece a los accesos a memoria**. El costo temporal de `bswap` también es considerable.

Las observaciones sobre `vpermq`, `vpinsrq`, `vpextrq` y `bswap` constituyen **evidencia fuerte a favor** de la hipótesis de que el **bajo rendimiento** de `cracker-loaded-p` se debe al **costo de las instrucciones utilizadas**. Se concluye, por lo tanto, que **guardar *P* en los registros YMM no es beneficioso**, por lo menos si la forma de aislar las subclaves es con `vpextrq` y `vpinsrq`; dadas las dependencias a nivel 64 bits involucradas en bcrypt, es poco probable que esto se pueda realizar de otra manera.

## 5.9. pbcrypt sin la instrucción más costosa

Durante el desarrollo del experimento anterior, se encontraron varios usos de la instrucción `vpermq` en `bcrypt-parallel.asm`. Dado el alto costo temporal de dicha instrucción, se elaboró la hipótesis de que **el uso de `vpermq` afecta negativamente el rendimiento de `cracker-parallel`**. Para ponerla a prueba, se implementó una variante de pbcrypt que se encuentra en el archivo `bcrypt-parallel-no-vpermq.asm`; el *cracker* asociado a esta implementación obviamente se llama `cracker-parallel-no-vpermq`.

En la primera implementación de pbcrypt, ciertas partes del cifrado se hacen manteniendo las **mitades izquierdas y derechas de los bloques** en un **mismo registro YMM** y **separándolas en dos registros XMM** cuando es necesario; en esta última acción, `vpermq` se usa para acceder a los 128 bits superiores del YMM. Esta decisión se debe a que durante el desarrollo del código, no se conocía la diferencia entre el costo de `vpermq` y el de escribir a la memoria principal, y se supuso que era conveniente ahorrar instrucciones `vmovdqa` (o `movdqa`) escribiendo 256 bits en lugar de 128.

[9] Macro `ROTATE_128`.
[10] Una *quadword*, o *qword*, corresponde a dos *dwords*, o sea 64 bits.

Como se puede ver en la figura 19, esta suposición era errada, y una sola ejecución de `vpermq` tarda más que dos `vmovdqa` (ya sean de lectura o de escritura). En vista de estos hechos, en la nueva variante de pbcrypt se optó por **mantener las mitades izquierdas en un XMM y las mitades derechas en otro**, realizando siempre **lecturas y escrituras de 128 bits**. Para comparar el rendimiento de `cracker-parallel-no-vpermq` con el de los *crackers* previamente desarrollados, se midió su tiempo de ejecución con los mismos datos de entrada usados en los experimentos ya hechos.

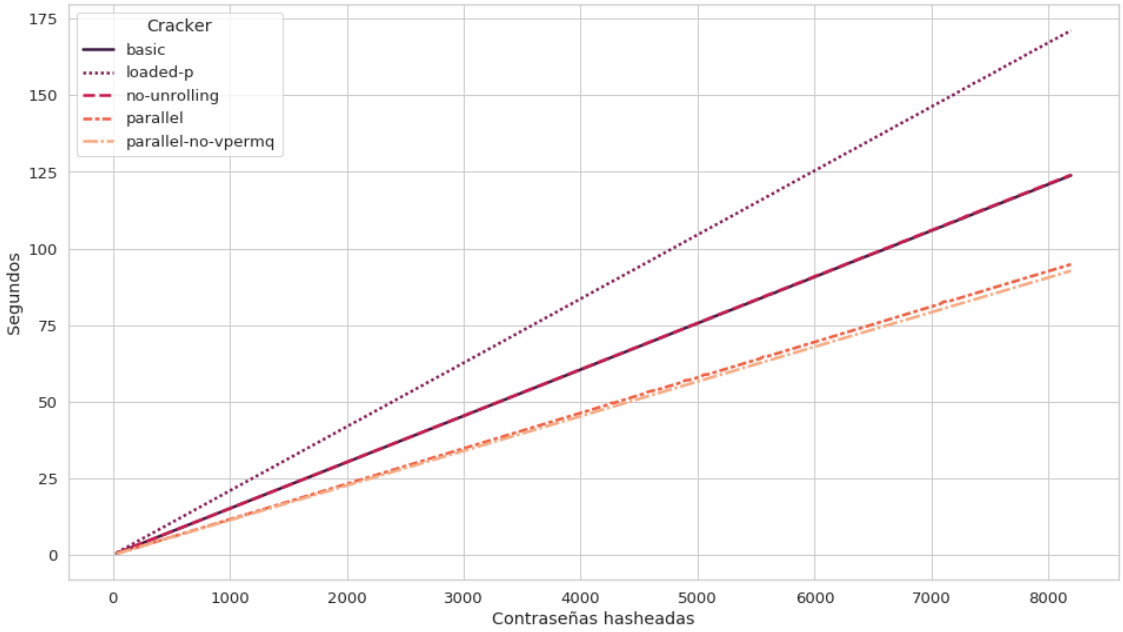


Figura 20: Rendimiento de los *crackers* para *wordlists* de longitudes 32 a 8192 y contraseñas de 13 bytes. Costo 8.

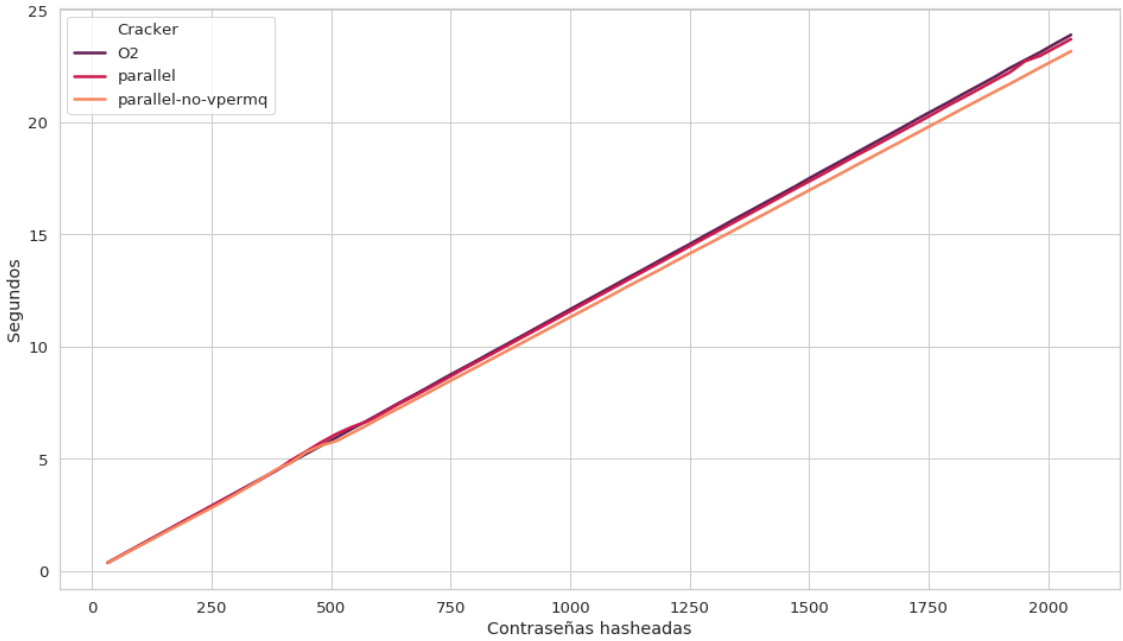


Figura 21: Rendimiento de los *crackers* para *wordlists* de longitudes 32 a 2048 y contraseñas de 13 bytes. Costo 8.

Las figuras 20 y 21 evidencian que eliminar los usos de `vpermq` resultó en una **leve mejora en el rendimiento de pbcrypt**. La diferencia entre su rendimiento y el del código C compilado con `-O2` sigue sin ser muy amplia, pero de todas formas es más significativa que la observada entre el primer *cracker* paralelo y `cracker-openbsd-O2`.

Una forma de implementar pbcrypt sin `vpermq` que no fue estudiada en este trabajo es reemplazar los usos de esta instrucción por `vextracti128`. De esta manera, en vez de acceder a los 128 bits más altos permutando *quadwords* en el mismo registro YMM, éstos se extraen a un registro XMM. Esta técnica permitiría hacer accesos a memoria de 256 bits sin el *overhead* asociado a `vpermq`, pero dado que no fue probada, no puede descartarse la posibilidad de que `vextracti128` también afecte negativamente el rendimiento.

## 5.10. pbcrypt doble

Las modificaciones al código de pbcrypt para operar siempre con las mitades izquierdas y derechas separadas permitieron adaptarlo fácilmente para trabajar con **registros YMM en lugar de XMM** y **datos de ocho contraseñas en cada uno**. El resultado de esta modificación es el algoritmo **pbcrypt doble** presentado en la subsección 3.2. La mayor parte de su código fuente es casi igual al de pbcrypt sin `vpermq`, pero con registros más anchos; la única diferencia cualitativa que tiene es que cuando lee *dwords* de ocho contraseñas, usa dos registros XMM e inserta uno en la parte superior del otro con la instrucción `vinserti128`, debido a que `pinsrq` y `vpinsrq` –las únicas instrucciones para insertar bytes en un registro multimedia– no aceptan operandos YMM (ver macro `READ_4_KEY_BYTES_PARALLEL_DOUBLE`).

Al igual que en el experimento anterior, los datos de entrada utilizados son los mismos que se usaron para experimentos previos, puesto que se compararon las mediciones obtenidas de pbcrypt doble con otras ya realizadas.

### 5.10.1. *Wordlist* creciente

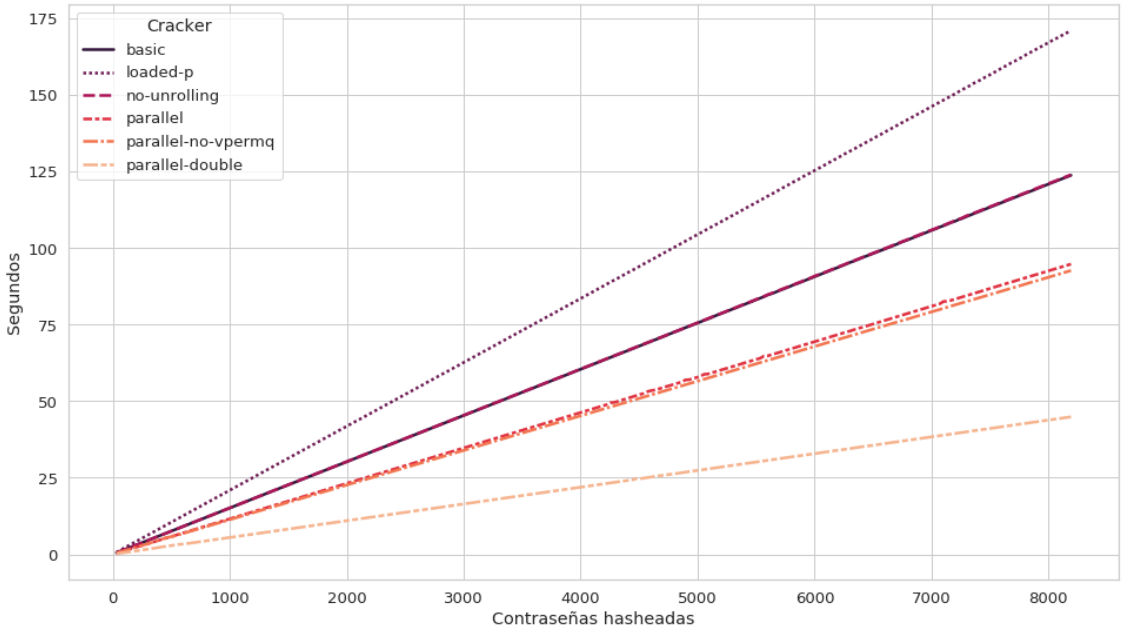


Figura 22: Rendimiento de los *crackers* para *wordlists* de longitudes 32 a 8192 y contraseñas de longitud 13. Costo 8.

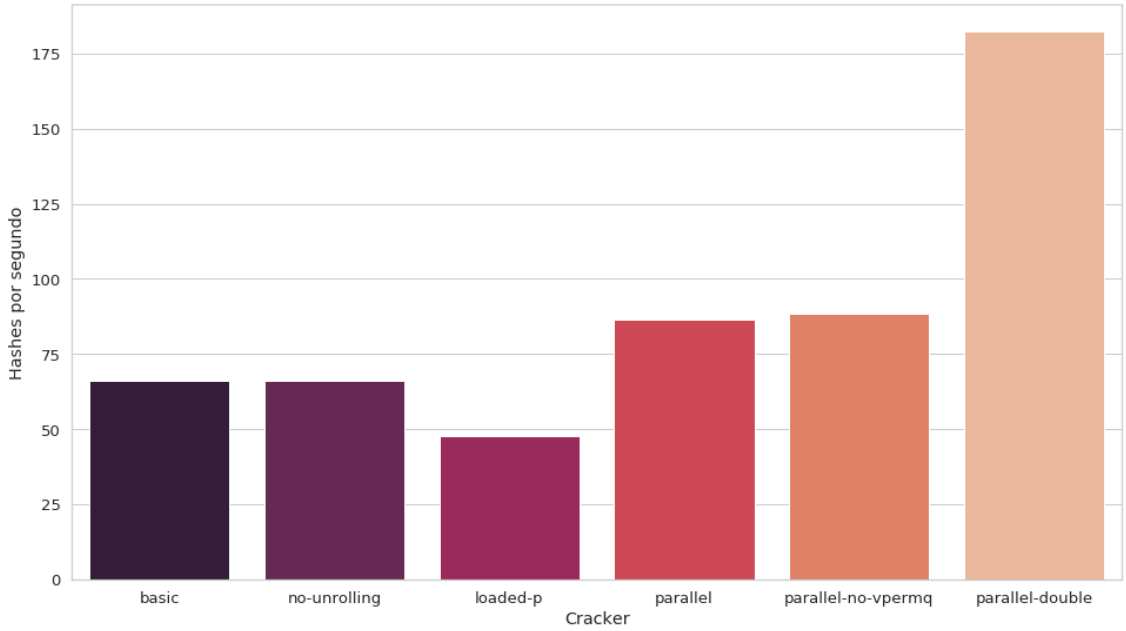


Figura 23: Hashes promedio por segundo. Costo 8.

Los resultados exhibidos en las figuras 22 y 23 evidencian que **pbcrypt doble es mucho más rápido que pbcrypt simple**. Mientras que la **tasa de *hashes* por segundo** del primer *password cracker* paralelo es de 86.3662 [11], la tasa del *cracker* paralelo doble es de 182.1356. Esto implica que la velocidad del *cracker* con ocho contraseñas simultáneas es **más del doble** que la del *cracker* con cuatro; concretamente, se dio una **mejora del 110 %**. La tasa de *hashes* por segundo de pbcrypt doble incluso **supera por un 106 %** a la de pbcrypt simple sin `vpermq`. Vale la pena contrastar esto con la mejora relativamente modesta de pbcrypt simple con respecto a bcrypt básico, que es del 33 %. A la vez, pbcrypt doble es **un 175 % más rápido** que bcrypt básico.

Resulta interesante que al procesar el doble de datos por vez, el nuevo algoritmo no sólo alcanza la **velocidad doble teórica**, sino que **la supera**. En base a los experimentos anteriores (en particular el del apartado 5.8), hay evidencia de que la **eliminación de `vpermq`** en pbcrypt doble influye en esta mejora [12], aunque sea ligeramente.

### 5.10.2. Costo

En el experimento de la subsección 5.3 se vio que a pesar de que pbcrypt simple es notablemente más veloz que bcrypt básico, el factor constante por el que mejora su complejidad temporal no introduce diferencias cualitativas muy importantes frente al crecimiento exponencial del tiempo de procesamiento en función del parámetro $C$. Dada la escala de la mejora obtenida para contraseñas hasheadas con costo 8, se decidió analizar el **rendimiento de pbcrypt doble para costos de 4 a 16**, tal como se había hecho para los cuatro *crackers* iniciales, y comparar los resultados con los obtenidos en el experimento correspondiente. Nuevamente, ningún incremento en la *performance*, por grande que sea, implicaría que la complejidad temporal del *key setup* deje de ser exponencial, pero en la práctica, podría tener un impacto muy positivo.

[11] La tasa de *hashes* por segundo fue calculada dividiendo la longitud de cada *wordlist* por el tiempo que llevó procesarla, y posteriormente promediando todos estos valores.

[12] Por supuesto que este razonamiento sólo se aplica a la primera implementación de pbcrypt.

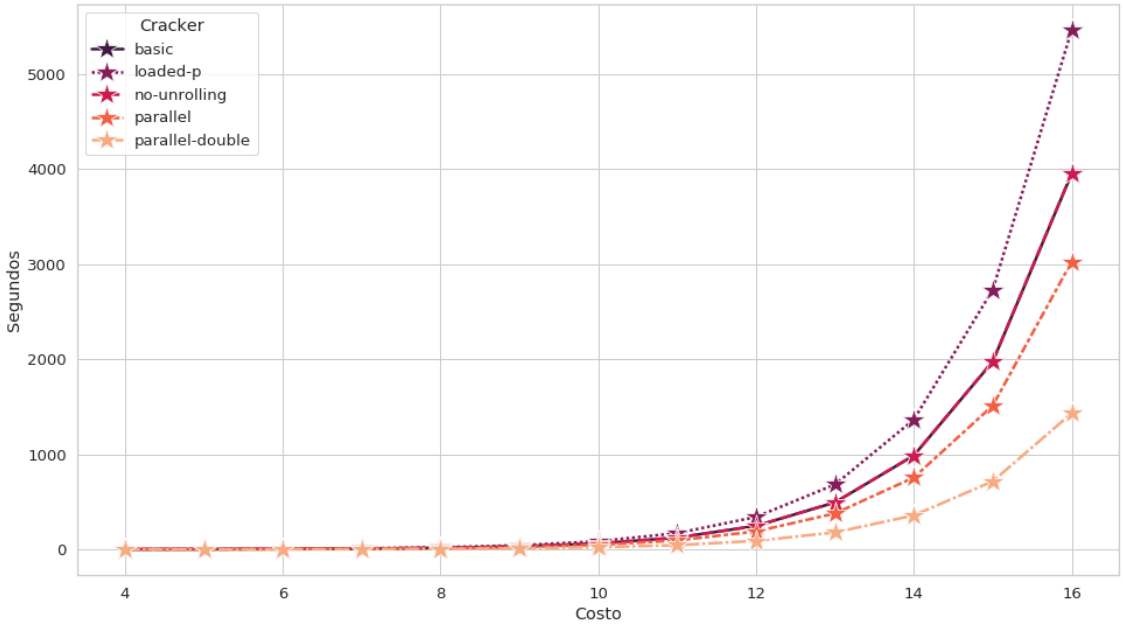


Figura 24: Rendimiento de los *crackers* para costos de 4 a 16. *Wordlist* de 1024 contraseñas.

La figura 24 indica que **pbcrypt doble es un éxito rotundo**. `cracker-parallel-double` **tarda el 47 %** de lo que tarda `cracker-parallel` en procesar la *wordlist* para una contraseña hasheada con costo 16, lo cual equivale a una **mejora del 111 %** (consistente con la observada en el experimento anterior). Obviamente, el incremento de velocidad es aún mayor con respecto al del *cracker* básico: proporcionalmente, `cracker-parallel-double` tardaría aproximadamente 16 horas en hacer lo que al básico le llevaría dos días. Ni siquiera el *overhead* asociado a procesar los datos de entrada para cumplir con los requerimientos de pbcrypt doble debería tener un efecto importante sobre esta ventaja, aunque el análisis de este fenómeno queda pendiente para trabajos futuros.

# 6. Conclusiones

A pesar de que bcrypt haya sido diseñado para ser imposible de vectorizar en una única entrada, el hecho de que los *password crackers* necesiten cifrar **muchas contraseñas** introduce la **posibilidad de paralelización a nivel datos**. El impacto de la **vectorización** en el rendimiento de los *crackers* desarrollados para este trabajo es **moderadamente positivo** cuando se opera con **cuatro claves a la vez** y **muy positivo** cuando se opera con **ocho a la vez**, lo cual indica que **tener registros más anchos** para manejar **más datos en paralelo** potencialmente da lugar al desarrollo de **variantes aún más rápidas**.

En ciertos casos de uso, las extensiones AVX y SSE pueden mejorar el rendimiento de los programas al ahorrar accesos a la memoria principal, pero **sustituir lecturas y escrituras por instrucciones SIMD no siempre es beneficioso**. Esto se pudo ver claramente en el **mal rendimiento del *cracker* con las subclaves cargadas** y en la comparación entre los costos de las diversas instrucciones. **Medir el tiempo de ejecución** de instrucciones poco ortodoxas como `vpermq` y `vpextrq` **antes de desarrollar programas** es una **buena práctica**; si se hubiera hecho este análisis en una etapa temprana del trabajo, la variante `bcrypt-loaded-p` jamás habría sido desarrollada.

# A. Especificaciones del sistema

```
[cat@eva-01 ~]$ uname -a
Linux eva-01 4.9.183-1-MANJARO #1 SMP PREEMPT Sat Jun 22 09:48:02 UTC 2019 x86_64
↪ GNU/Linux

[cat@eva-01 ~]$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          4
On-line CPU(s) list:             0-3
Thread(s) per core:              1
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           158
Model name:                      Intel(R) Core(TM) i5-7600 CPU @ 3.50GHz
Stepping:                        9
CPU MHz:                         899.993
CPU max MHz:                     4100,0000
CPU min MHz:                     800,0000
BogoMIPS:                        7010.00
Virtualization:                  VT-x
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        6 MiB
NUMA node0 CPU(s):               0-3
Vulnerability L1tf:              Mitigation; PTE Inversion; VMX conditional cache
↪ flushes, SMT disabled
Vulnerability Mds:               Mitigation; Clear CPU buffers; SMT disabled
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled
↪ via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, IBPB
↪ conditional, IBRS_FW, STIBP disabled, RSB filling
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
↪ pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
↪ syscall nx pdpe1gb rdtscp lm constant
                                 _tsc art arch_perfmon pebs bts rep_good nopl
                                 ↪ xtopology nonstop_tsc aperfmperf pni
                                 ↪ pclmulqdq dtes64 monitor ds_cpl vmx smx est
                                 ↪ tm2 ssse3 sdbg fma cx16 xtpr pd
                                 cm pcid sse4_1 sse4_2 x2apic movbe popcnt
                                 ↪ tsc_deadline_timer aes xsave avx f16c rdrand
                                 ↪ lahf_lm abm 3dnowprefetch invpcid_single
                                 ↪ ssbd ibrs ibpb stibp kaiser t
```

```
                                    pr_shadow vnmi flexpriority ept vpid fsgsbase
                                    ↪  tsc_adjust bmi1 hle avx2 smep bmi2 erms
                                    ↪  invpcid rtm mpx rdseed adx smap clflushopt
                                    ↪  intel_pt xsaveopt xsavec xge
                                    tbv1 xsaves dtherm ida arat pln pts hwp
                                    ↪  hwp_notify hwp_act_window hwp_epp md_clear
                                    ↪  flush_l1d

[cat@eva-01 ~]$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-pc-linux-gnu/9.1.0/lto-wrapper
Target: x86_64-pc-linux-gnu
Configured with: /build/gcc/src/gcc/configure --prefix=/usr --libdir=/usr/lib
↪  --libexecdir=/usr/lib --mandir=/usr/share/man --infodir=/usr/share/info
↪  --with-bugurl=https://bugs.archlinux.org/
↪  --enable-languages=c,c++,ada,fortran,go,lto,objc,obj-c++ --enable-shared
↪  --enable-threads=posix --with-system-zlib --with-isl --enable-__cxa_atexit
↪  --disable-libunwind-exceptions --enable-clocale=gnu --disable-libstdcxx-pch
↪  --disable-libssp --enable-gnu-unique-object --enable-linker-build-id
↪  --enable-lto --enable-plugin --enable-install-libiberty
↪  --with-linker-hash-style=gnu --enable-gnu-indirect-function --enable-multilib
↪  --disable-werror --enable-checking=release --enable-default-pie
↪  --enable-default-ssp --enable-cet=auto
Thread model: posix
gcc version 9.1.0 (GCC)

[cat@eva-01 ~]$ nasm -v
NASM version 2.14.02 compiled on Jan 22 2019
```

# Referencias


[1] *Cryptographic algorithm Blowfish*. URL: http://cryptowiki.net/index.php?title=Cryptographic_algorithm_Blowfish.

[2] Nik Cubrilovic. *RockYou Hack: From Bad To Worse*. URL: https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/.

[3] Agner Fog. «Loops». En: *Optimizing subroutines in assembly language*. 1996. Cap. 12, págs. 88, 101-103.

[4] Agner Fog. «Optimizing memory access». En: *Optimizing subroutines in assembly language*. 1996. Cap. 12, págs. 81-88.

[5] Jonathan Katz y Yehuda Lindell. *Introduction to Modern Cryptography*. Taylor & Francis Inc, 2014. ISBN: 9781466570269.

[6] Matt Marx. *A Practical Guide to Cracking Password Hashes*. URL: https://labs.f-secure.com/archive/a-practical-guide-to-cracking-password-hashes/.

[7] David Mazières y Niels Provos. «A Future-Adaptable Password Scheme». En: *Proceedings of the FREENIX Track. USENIX Annual Technical Conference*. 1999.

[8] David Mazières y Niels Provos. «A Future-Adaptable Password Scheme». En: *Proceedings of the FREENIX Track. USENIX Annual Technical Conference*. 1999, pág. 7.

[9] David Mazières y Niels Provos. «A Future-Adaptable Password Scheme». En: *Proceedings of the FREENIX Track. USENIX Annual Technical Conference*. 1999, págs. 2, 6.

[10] Alfred J. Menezes, Paul C. van Oorschot y Scott A. Vanstone. *Handbook of Applied Cryptography*. Taylor & Francis Inc, 1997, págs. 251, 254. ISBN: 9780429466335.

[11] Alfred J. Menezes, Paul C. van Oorschot y Scott A. Vanstone. *Handbook of Applied Cryptography*. Taylor & Francis Inc, 1997, pág. 230. ISBN: 9780429466335.

[12] Alfred J. Menezes, Paul C. van Oorschot y Scott A. Vanstone. *Handbook of Applied Cryptography*. Taylor & Francis Inc, 1997, pág. 390. ISBN: 9780429466335.

[13] Philippe Oechslin. «Making a Faster Cryptanalytic Time-Memory Trade-Off». En: *Lecture Notes in Computer Science* 2729 (2003).

[14] *passwd(5) - OpenBSD Manual Pages*. URL: https://man.openbsd.org/passwd.5.

[15] Bruce Schneier. *The Blowfish Encryption Algorithm*. URL: https://www.schneier.com/academic/blowfish/.