מה זה באמת SHA-256

טביעת אצבע מתמטית שנכנסת בשישים וארבעה תווים ומשתנה כולה אם פסיק בודד בטקסט המקורי זז. מדוע אנחנו קוראים לזה חותם שעווה דיגיטלי.

הרעיון הפשוט מאחורי השם הטכני

דמיינו שקיימת מכונה עם חריץ אחד ומסך אחד. דרך החריץ אתם מכניסים טקסט: מילה, משפט, רומן שלם. על המסך מופיע, כעבור רגעים ספורים, רצף של בדיוק שישים וארבעה תווים. הרצף הזה, לקורא המקצועי אנו קוראים לו *hash* או *סיכום קריפטוגרפי*; לקורא הכללי, נוכל לקרוא לו לעת עתה טביעת אצבע מתמטית של הטקסט, כפי שטביעת אצבע היא של אדם.

אם תכניסו את אותו הטקסט פעמיים, המכונה תציג את אותה טביעת אצבע פעמיים. אם תכניסו טקסט מעט שונה — פסיק בודד שהוזז, אות גדולה שהפכה לקטנה — המכונה תציג טביעת אצבע שונה לחלוטין מהראשונה. לא דומה: שונה. שתי התכונות הללו יחד — דטרמיניזם ורגישות — הן הרעיון הפשוט. כל השאר ב-SHA-256 הוא המנגנון שגורם להן להתקיים היטב.

כדאי לומר כבר מהתחלה מה המכונה לא עושה. היא לא מצפינה את הטקסט. היא לא מסתירה אותו. היא לא שומרת אותו. המכונה מסתכלת על הטקסט, מחשבת את טביעת האצבע, ושוכחת מהטקסט. טביעת האצבע אינה מאפשרת לשחזר את הטקסט שיצר אותה; היא רק מאפשרת, בהינתן טקסט מועמד, לבדוק אם הוא תואם למקור או לא. לכן אנו אומרים שזהו סיכום *חד-כיווני*: הולכים, לא חוזרים.

Hash זה לא אותו דבר כמו הצפנה

הבלבול נפוץ וכדאי להבהיר אותו: הצפנה ו-hashing הן פעולות שונות. הצפנה מורכבת משינוי טקסט כך שרק בעל המפתח יוכל להחזיר אותו לצורתו המקורית. Hashing מורכב מיצירת טביעת אצבע של הטקסט שממנה לעולם לא ניתן לשחזר את הטקסט המקורי, לא עם מפתח ולא בלעדיו. הראשונה הפיכה לפי תכנון; השנייה בלתי הפיכה לפי תכנון.

התוצאה המעשית חשובה. כשאפליקציה אומרת «אנחנו שומרים את הסיסמה שלך מוצפנת», יש מישהו שיש לו את המפתח לפענח אותה — האפליקציה עצמה, בכל מקרה. כשאפליקציה אומרת «אנחנו שומרים את הסיסמה שלך ב-hash», האפליקציה עצמה לא יכולה לקרוא את הסיסמה המקורית גם אם הייתה רוצה; היא רק יכולה לבדוק אם מה שאתה כותב מייצר שוב את אותה טביעת אצבע. המודל השני, אם נעשה נכון, עדיף בהרבה על הראשון לאחסון סיסמאות. בהמשך נראה מדוע «נעשה נכון» דורש קצת יותר מאשר SHA-256 בלבד.

ארבע התכונות שהופכות hash קריפטוגרפי לשימושי

פונקציית hash הראויה לתואר *קריפטוגרפי* מקיימת ארבע תכונות:

1. **דטרמיניזם.** אותה קלט מייצר תמיד את אותה טביעת אצבע.
2. **אפקט המפולת (Avalanche).** שינוי קטן בקלט מייצר טביעת אצבע שונה לחלוטין, ללא דמיון נראה לעין לקודמת.
3. **עמידות בפני היפוך.** בהינתן טביעת אצבע, לא ניתן חישובית למצוא את הטקסט שיצר אותה.
4. **עמידות בפני התנגשויות (Collisions).** לא ניתן חישובית למצוא שני טקסטים שונים המייצרים את אותה טביעת אצבע.

«לא ניתן חישובית» אין פירושו «זה בלתי אפשרי מתמטית». פירושו שהעלות בזמן, אנרגיה וכסף כדי להשיג זאת עולה בסדרי גודל על סך כל כושר המחשוב הזמין באופן סביר. עבור SHA-256, הגבול הזה נמדד באלפי טריליוני שנים אפילו עבור הגישות האופטימיות ביותר עם חומרה ייעודית. מה שלמטרות המעשיות של הקורא, הוא זהה ל-«לא ניתן».

SHA-256, באופן ספציפי

השם אומר הכל. SHA הן ראשי התיבות של *Secure Hash Algorithm*: אלגוריתם hash מאובטח. המספר 256 מציין את גודל טביעת האצבע בביטים: מאתיים חמישים ושישה ביטים, כלומר שלושים ושניים בייטים, המוצגים בהקסדצימל הם שישים וארבעה התווים שהקורא כבר מזהה. התקן פורסם על ידי ה-NIST האמריקאי, הגוף המתקנן פונקציות מסוג זה, בשנת 2001 כחלק ממשפחת SHA-2; הגרסה הנוכחית של התקן, FIPS 180-4, היא משנת 2015.

למי שעדיין לא זוכר מהם ביטים ובייטים:

```
1 ביט     →  0 או 1          (מפסק: דלוק או כבוי)
1 בייט    →  8 ביטים         (256 שילובים אפשריים)
32 בייטים →  256 ביטים       (טביעת האצבע SHA-256)
```

המספר 256 בסוף השם אומר את גודל טביעת האצבע בביטים. בהקסדצימל — שיטת ספירה עם שישה עשר סמלים במקום עשרה — 256 הביטים הללו נכנסים בדיוק ב-64 תווים. אלו הם 64 התווים שאתה רואה בתחתית כל Cuaderno.

הממדים ראויים לרגע של מחשבה. מאתיים חמישים ושישה ביטים מאפשרים שתיים בחזקת מאתיים חמישים ושש ערכים שונים: מספר עם שבעים ושמונה ספרות עשרוניות, גדול בכמה סדרי גודל ממספר האטומים המוערך ביקום הנצפה. כל טקסט בעולם — כל ספר, כל דוא"ל, כל הודעה — נופל על אחד מהערכים הללו. ההסתברות ששני טקסטים שונים יתלכדו במקרה היא, למטרות מעשיות, בלתי ניתנת להבחנה מאפס.

איך זה נראה בקוד

ב-Zig, השפה שבה אנו כותבים את החלקים המחזיקים את Solo2, חישוב חותם SHA-256 של טקסט נראה כך:

```
const std = @import("std");

const texto = "Cuadernos Lacre";
var resumen: [32]u8 = undefined;
std.crypto.hash.sha2.Sha256.hash(texto, &resumen, .{});
```

זה עתה ביקשנו מהספרייה הסטנדרטית של Zig לחשב את ה-SHA-256 של הטקסט שבמרכאות. לאחר הקריאה, המשתנה *resumen* מכיל את שלושים ושניים הבייטים המרכיבים את החותם בצורתו הגולמית; כאשר הם מוצגים על המסך בהקסדצימל, אלו הם שישים וארבעה התווים המופיעים בתחתית מאמר זה. אם היינו משנים את *Cuadernos Lacre* ל-*Cuadernos lacre* — אות גדולה אחת פחות — החותם היה משתנה כולו. זוהי, בחמש שורות, התכונה המרכזית המחזיקה את כל השאר. למי שרוצה לראות איך זה עובד מבפנים, בסוף המאמר כללנו גרסה קריאה של האלגוריתם עם הערות צעד אחר צעד.

מדוע אנחנו קוראים לזה חותם שעווה

בתכתובת האירופית של המאות החמש עשרה עד התשע עשרה, חותם השעווה סגר את המכתב. טיפת שעווה מותכת, חותם שנלחץ עליה, והמכתב נשאר מסומן בצורה שאינה ניתנת לשחזור. זה לא הגן על התוכן מפני המציצן הנחוש — ניתן היה לקרוא את הנייר מול האור, ניתן היה לשבור את השעווה — אבל זה העיד עליו. כל שינוי בסגירה היה גלוי לנמען עוד לפני פתיחת הנייר. חותם השעווה לא מנע את הנזק; הוא הכריז עליו.

ה-SHA-256 של גוף כל Cuaderno ממלא את אותו התפקיד בגרסתו הדיגיטלית. אם מילה בודדת במאמר תשתנה בין רגע הפרסום לרגע שבו אתה קורא אותו, החותם ההקסדצימלי בתחתית הטקסט כבר לא יתאים ל-SHA-256 של הטקסט שלפניך. כל קורא עם חמש שורות קוד יכול לבדוק זאת. הפרסום אינו יכול לשכתב את ההיסטוריה שלו מבלי שהחותם יסגיר אותו. הוא אינו מגן מפני נזק; הוא הופך אותו לניתן לאימות.

מה hash הוא לא

ארבעה שימושים מתבקשים לעיתים מ-SHA-256 שאינם בתחום אחריותו:

1. **הצפנה.** Hash מסכם; הוא לא מסתיר. אם אתה רוצה שהטקסט לא יהיה קריא, אתה צריך להצפין אותו, לא לעשות לו hash.
2. **אימות המחבר.** Hash לא אומר מי כתב את הטקסט, רק איזה טקסט עבר hashing. כדי לשייך מחבר דרושה חתימה קריפטוגרפית מעל ה-hash, לא ה-hash בלבד.
3. **אחסון סיסמאות.** כאן יש מלכודת שכדאי להבין. SHA-256 תוכנן להיות מהיר מאוד — מה שטוב לדברים רבים, אבל רע לזה. תוקף עם חומרה ייעודית יכול לנסות מיליארדי סיסמאות בשנייה מול hash SHA-256 hasta שימצא את שלך. כדי לשמור סיסמאות יש להשתמש בפונקציות גזירת מפתח איטיות בכוונה כמו Argon2, scrypt או bcrypt, בשילוב עם *מלח* (salt - נתון אקראי ייחודי לכל משתמש, המונע משני אנשים עם אותה סיסמה לקבל את אותו ה-hash).
4. **קריאת ה-hash כמזהה המחבר.** זה לא. Hash מזהה את התוכן. אם שני אנשים עושים hash למילה *שלום* עם SHA-256, שניהם מקבלים את אותו הסיכום — וזוהי התכונה המרכזית, לא פגם: אם אלו היו סיכומים שונים, לא היינו יכולים לבדוק התאמה בין מה שפורסם למה שהתקבל.

איפה מופיע SHA-256 בחיי היומיום שלך

למרות שאינך רואה זאת, SHA-256 מחזיק חלק גדול ממה שאתה משתמש בו מדי יום באינטרנט. שרשרת הבלוקים של ביטקוין נבנית על ידי שרשור SHA-256 של כל בלוק לבלוק הבא; שינוי בלוק מהעבר מחייב חישוב מחדש של כל השרשרת שבאה אחריו. Git, המערכת שבה מנהלים גרסאות של קוד בחצי מהעולם, מזהה כל אישור (commit) לפי ה-SHA-256 (בגרסאות אחרונות) או לפי קודמו SHA-1 (בגרסאות ישנות יותר) של כל תוכנו. תעודות HTTPS המאמתות את זהות האתר כשאתה נכנס אליו נושאות טביעת אצבע SHA-256 משויכת. הורדות תוכנה מלוות לעיתים קרובות ב-SHA-256 שפורסם על ידי המפתח כדי שתוכל לוודא שהקובץ לא שונה בדרך. וכפי שאמרנו, בתחתית כל Cuaderno Lacre.

לקורא המקצועי

ארבע תזכורות תפעוליות למי שמחליט או מבקר מערכות:

1. Hash אינו הצפנה. אם ספק מתבלבל בין שני המונחים בתיעוד הטכני שלו, כדאי לשאול למה הוא מתכוון בדיוק.
2. כדי לאחסן סיסמאות לעולם אין להשתמש ב-SHA-256 בלבד. SHA-256 מהיר מדי למשימה זו (ראה סעיף 3 ב-*מה hash הוא לא*). התקן הנוכחי הוא **Argon2id**: איטי לפי תכנון, ניתן להגדרה לפי יכולת השרת, בשילוב עם *מלח* אקראי שונה לכל משתמש.
3. עבור שלמות מסמכים — חוזים, תיקים, קבצים — SHA-256 נותר תקן הייחוס. זהו התקן שבו משתמשים נותני שירותי חותם זמן מוסמכים באיחוד האירופי.
4. לשימור לטווח ארוך (עשורים) כדאי לחשב ולארכב גם SHA-3 או SHA-512 לצד ה-SHA-256; הזהירות הקריפטוגרפית ממליצה לא להסתמך על פונקציה בודדת עבור ארכיונים של מאה שנה.

מבחינה טכנית, מבנה איטרטיבי זה — שבו המצב המצטבר נשמר בין בלוקים של קלט — מוכר כבניית **Merkle-Damgård**, התבנית שעליה מבוססות פונקציות ה-hash הקלאסיות SHA-1, SHA-2 (כולל SHA-256) ורבות אחרות. SHA-3, לעומת זאת, זונחת את Merkle-Damgård לטובת ארכיטקטורה שונה הנקראת *ספוג* (sponge).

כיצד פועל SHA-256, צעד אחר צעד, בשפה פשוטה

דמיינו שהקמתם את מסלול הדומינו המורכב ביותר בעולם: אלפי אבנים, עשרות הסתעפויות, גשרים מכניים ורמפות שחוצות את החדר כולו, מונחות בזהירות אבן אחר אבן.

אם תתנו נקישה לאבן הראשונה, השרשרת תיפול ברצף מדויק וניתן לשחזור. אותה הקמה, אותה נקישה ראשונית ← תבנית סופית זהה של אבנים שנפלו, שוב ושוב.

הנה החלק המעניין: הזיזו **אבן אחת בלבד** חצי סנטימטר הצידה לפני שמתחילים, ונקשו שוב. רמפה שהייתה אמורה להיות מופעלת נשארת דוממת, גשר לא נופל, הסתעפות שונה מופעלת. התבנית הסופית של האבנים על הרצפה בלתי ניתנת לזיהוי לחלוטין בהשוואה לראשונה.

SHA-256 הוא המסלול הזה מבחינה מתמטית. הטקסט שאתם כותבים הוא המיקום הראשוני של האבנים. האלגוריתם הוא הנקישה שמשחררת את המפל. והתוצאה הסופית — מה שאנו מכנים *hash* — היא תצלום סטילס של הרצפה כשהכל נעצר. שנו פסיק אחד בטקסט המקורי, והתמונה תהיה שונה בתכלית. פשוט כל כך, ודרסטי כל כך.

שלב 1. תרגום הטקסט לאבני דומינו בינאריות. מחשבים לא מבינים אותיות; הם מתרגמים אותן תחילה למספרים (ASCII) ואת המספרים לבינארי (אחדים ואפסים). כל אות הופכת ל-8 אבנים לבנות או שחורות: *A* היא `01000001`, *B* היא `01000010`, והרווח הוא `00100000`. הטקסט כולו שלכם — מילה, חוזה, רומן — הופך לשורה ארוכה של אבנים לבנות ושחורות.

שלב 2. מילוי עד לגודל סטנדרטי. המסלול מעבד את השורה *במקטעים* של 512 אבנים בדיוק. אם ההודעה שלכם לא מגיעה לכפולות של 512, נוספת אבן סימון (בעלת הערך `10000000`) מיד אחרי הטקסט ולאחר מכן אפסים עד להשלמת המקטע. 64 המיקומים האחרונים של כל מקטע נשמרים לרישום האורך המקורי של הטקסט. כך המסלול תמיד יודע היכן הסתיים התוכן האמיתי והיכן התחיל המילוי.

שלב 3. הצבת שמונה אבני המאסטר. לפני שמתחילים, אנו מניחים על השולחן **שמונה אבני מאסטר** במיקום ראשוני מדויק. שמונה אבנים אלו אינן סוד: ערכן הראשוני נקבע לפי כלל מתמטי ציבורי (השורשים הריבועיים של שמונת המספרים הראשוניים הראשונים — 2, 3, 5, 7, 11, 13, 17, 19 — והביטים הראשונים של החלק העשרוני של כל שורש). כולם, בכל פינה בעולם, מתחילים עם אותן שמונה אבני מאסטר באותו מיקום. גורלן להידחף ולהשתנות על ידי המפל.

שלב 4. המפל הגדול: שישים וארבעה סבבי דחיפות. כאן מתחילה ההצגה. המקטע הראשון של 512 אבנים מהטקסט שלכם מתנגש בשמונה אבני המאסטר. אך הן אינן נופלות בבת אחת: המנגנון מבצע **שישים וארבעה סבבים רצופים**. בכל סבב הוא מבצע שלוש פעולות עם האבנים:

- **הקרוסלה** (רוטציה). האבנים נעות במעגל: אלו שמימין עוברות לשמאל. אף אבן לא הולכת לאיבוד ולא נוספת; הן פשוט מסודרות מחדש בסיבוב שלם של הקרוסלה. זו דרך זולה והפיכה לחלוקה מחדש של המידע.
- **המשפך הלוגי** (XOR). האבנים עוברות דרך משפך שמשווה אותן זוגות-זוגות: אם שתיהן באותו צבע, יוצאת אבן לבנה; אם הן שונות, יוצאת אבן שחורה. זו הפעולה הפשוטה ביותר בלוגיקה בינארית, אך בשילוב עם סיבובי הקרוסלה היא הופכת לעוצמתית ביותר לערבוב מידע מבלי לאבדו.
- **הגלישה** (חיבור מודולרי). התוצאה מחוברת עם *אבן דחיפה קבועה* שנלקחת מרשימה ציבורית של שישים וארבעה קבועים (השורשים השלישיים של שישים וארבעה המספרים הראשוניים הראשונים). אם החיבור יוצר אבנים נוספות שאינן נכנסות בשטח המוקצב של 32 אבנים, האבנים העודפות הללו נזרקות. לשולחן יש מקום ל-32 אבנים בלבד, לא אחת יותר.

בסיום סבב שישים וארבע, כל אחת מהאבנים במקטע הטקסט שלכם השפיעה על המיקום של שמונה אבני המאסטר. אנרגיית הדחיפה עברה דרך המסלול כולו.

שלב 5. הוספת המקטע הבא (מבלי לאפס). אם הטקסט שלכם היה ארוך ונותר מקטע נוסף של 512 אבנים לעיבוד, **המסלול לא מתאפס**. שמונה אבני המאסטר נשארות כפי שהשאיר אותן המפל הראשון, והמקטע השני משוחרר לעברן כדי להפעיל שישים וארבעה סבבים נוספים. זה כמו להוסיף חדר חדש מלא באבני דומינו בסוף זה שזה עתה נפל: אי-הסדר של הראשון קובע לחלוטין כיצד ייפול השני.

שלב 6. צילום התמונה הסופית. כשלא נותרו עוד מקטעים לעיבוד, המפל נעצר. אנו מסתכלים על המיקום הסופי שבו נותרו שמונה אבני המאסטר. אנו מתרגמים את התצורה שלהן לקוד של אותיות ומספרים בשיטה ההקסדצימלית. התוצאה היא מחרוזת של 64 תווים בדיוק: זהו חותם ה-SHA-256 שלכם.

ארבע תכונות נובעות מאליהן מהאופן שבו המסלול בנוי:

1. **דטרמיניזם.** אותו טקסט מייצר תמיד את אותה תמונה סופית, בכל מחשב בעולם. אפס אקראיות, אפס הפתעות.
2. **אפקט המפל.** פסיק שנוסף, אות גדולה שהשתנתה, סימן ניקוד שנשכח: התמונה הופכת לבלתי ניתנת לזיהוי לחלוטין. זוהי הרגישות הקיצונית שכבר תיארנו בהתחלה.

3. **כיוון אחד בלבד.** בהינתן התמונה הסופית, לא ניתן לשחזר את הטקסט המקורי. הרוטציות, המשפכים והגלישות משמידים את כל המידע הכיווני לגבי *מהיכן הגיע כל ביט* ושומרים רק על *מה שסוכם בסך הכל*.
4. **עמידות בפני התנגשויות.** בעשרים וחמש שנות קריפטואנליזה ציבורית, איש לא הצליח למצוא שני טקסטים שונים שהתמונות הסופיות שלהם זהות. והקושי לעשות זאת נמצא מחוץ להישג היד החישובי של כל ציוויליזציה שניתן להעלות על הדעת.

נספח הקוד שבא בהמשך מיישם בדיוק את ששת השלבים הללו ב-Zig. כעת תוכלו לקרוא אותו כשאתם יודעים מה המשמעות של כל פעולת ביטים, במקום לקבל את המניפולציות בעיניים עצומות.

מילון מונחים טכני

לקורא המעוניין להבין מה עושה כל פעולה. דלגו על כך בחופשיות: המאמר מובן גם בלעדיו.

ASCII ו-**Unicode — כיצד אותיות הופכות למספרים.** מחשבים לא רואים אותיות; הם רואים מספרים. תקן שנקרא **ASCII** (*American Standard Code for Information Interchange*, משנת 1963) מקצה לכל תו במקלדת מספר ספציפי: *A* היא 65, *B* היא 66, *a* היא 97, *0* הוא 48, הרווח הוא 32, הפסיק הוא 44. מערכות מודרניות מרחיבות זאת עם **Unicode**, המקצה מספר לכל תו בכל אלף-בית בעולם: קירילית, ערבית, סינית, יפנית ואפילו אמוג'י. כשאתם כותבים תו או פותחים קובץ טקסט, המחשב קורא את המספר שמאחורי הקלעים, לא את הצורה על המסך. SHA-256 עובד על המספרים הללו, ומתייחס לכל טקסט כרצף ארוך של ספרות. לכן הוא יכול לחתום על מאמר בספרדית, שיר ביפנית וקובץ בינארי באותו אלגוריתם.
XOR — המשווה ביט אחר ביט. XOR (נהגה «*אקס-אור*», מהביטוי האנגלי *exclusive or*, «או מוציא») הוא אחת הפעולות הפשוטות ביותר שמחשב יכול לבצע עם שני מספרים בינאריים. הוא משווה שני ביטים מיקום אחר מיקום ומחזיר: **1** אם בדיוק אחד מהשניים הוא 1 (אחד מהם אך לא שניהם), **0** אם שניהם זהים (שניהם 0 או שניהם 1). דוגמה: XOR של `1010` ו-`1100` הוא `0110`. יש לו תכונה ראויה לציון: הוא הפיך — אם מבצעים XOR פעמיים עם אותו מפתח, חוזרים למקור. לכן הוא סוס העבודה של הקריפטוגרפיה: הוא מערבב ביטים מבלי לאבד מידע, אך התוצאה אינה חושפת דבר על הקלטים אם אינכם יודעים אחד מהם.
הקסדצימלי — ספירה בבסיס 16. כמעט כל המספרים בחיי היומיום משתמשים בעשר ספרות (0-9). השיטה ההקסדצימלית משתמשת בשש-עשרה: הספרות הרגילות 0-9 פלוס שש אותיות המייצגות את הערכים הבאים: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. למה שש-עשרה? כי מחשבים חושבים בקבוצות של ארבעה ביטים, וארבעה ביטים יכולים לייצג בדיוק שש-עשרה ערכים שונים — כך, תו הקסדצימלי אחד מתאים בצורה נקייה לארבעה ביטים. טביעת אצבע של SHA-256 היא באורך 256 ביטים, שהם בדיוק **64 תווים הקסדצימליים**. אילו היינו כותבים אותה בדצימלי רגיל, היא הייתה תופסת כ-78 ספרות והייתה פחות נוחה. הבחירה היא אסתטית וקומפקטית; המספר שמאחורי הקלעים הוא אותו הדבר.
רוטציית ביטים — הקרוסלה הבינארית. דמיינו שורה של שבע נורות, חלקן דולקות (1) וחלקן כבויות (0): `1 0 1 1 0 0 1`. רוטציה ימינה במיקום אחד מורכבת מלקיחת הנורה הימנית ביותר, העברתה לקצה השמאלי והזזת השאר מקום אחד ימינה: `1 1 0 1 1 0 0`. אף נורה לא הולכת לאיבוד ולא נוספת: הן פשוט רוקדות במעגל. SHA-256 משתמש ברוטציית ביטים מאות פעמים בכל חישוב; זו דרך זולה וללא אובדן לחלוקה מחדש של המידע בתוך המצב.
קבועי *«nothing-up-my-sleeve»* — מדוע הם מגיעים ממספרים ראשוניים. שמונה אבני המאסטר ושישים וארבעה קבועי הסבב של SHA-256 לא נבחרו באקראי. הם מגיעים מהשורשים הריבועיים והשלישיים של המספרים הראשוניים הראשונים. למה? כי המעצבים שלהם רצו קבועים «*ללא שום דבר בשרוול*» (*«nothing-up-my-sleeve»*): ערכים שכל אחד יכול לאמת את מקורם. אם מישהו היה אומר לכם «*סמכו עליי: השתמשו במספר אקראי זה של 32 ביט*», הייתם חושדים בצדק בחולשה נסתרת או בדלת אחורית. אך כל אחד עם מחשבון יכול לבדוק ש-32 הביטים הראשונים של השורש הריבועי של 2 הם `0x6a09e667`. הערכים הם מתמטיים, ציבוריים וניתנים לשחזור: אף מלכודת נסתרת אינה יכולה להשתרבב למתכון.

נספח: SHA-256 בקוד קריא

נספח זה מיועד לקורא שרוצה לראות את האלגוריתם מבפנים. זהו מימוש דידקטי ב-Zig העוקב אחר מפרט FIPS 180-4. זו אינה הגרסה שבה משתמשת Solo2 — הגרסה האמיתית נמצאת ב-`std.crypto.hash.sha2.Sha256` של הספרייה הסטנדרטית של Zig, אופטימלית ומבוקרת. אבל האלגוריתם הוא אותו דבר: מה שאתה רואה כאן הוא, שלב אחר שלב, מה שקורה כאשר אותה קריאה של חמישה תווים מבצעת את עבודתה.

```zig
;const std = @import("std")

.SHA-256 — implementación didáctica //
   Sigue la especificación FIPS 180-4. Prioriza la claridad sobre la //
  ,velocidad y la robustez frente a entradas hostiles. Para producción //
      .usa std.crypto.hash.sha2.Sha256, que está optimizada y auditada //

 H0: las ocho palabras del estado inicial. Primeros 32 bits de la parte //
       fraccionaria de las raíces cuadradas de los primeros ocho primos //
                                          .(19 ,17 ,13 ,11 ,7 ,5 ,3 ,2) //
                                                  }const H0 = [_]u32
                                ,0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a
                                ,0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
                                                                        ;{

   K: 64 constantes de ronda. Primeros 32 bits de la parte fraccionaria //
                                .de las raíces cúbicas de los primeros 64 primos //
                                                    }const K = [_]u32
8a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5
07aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174
9b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da
3e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967
b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85
bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070
a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3
8f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
                                                                        ;{

                                    .Rotación circular a la derecha de un u32 //
                                            } inline fn rotr(x: u32, n: u5) u32
                                           ;return std.math.rotr(u32, x, n)
                                                                         {

                                    .Lee 4 bytes consecutivos como un u32 big-endian //
                                                  } inline fn readU32(b: []const u8) u32
;return @as(u32, b[0]) << 24 | @as(u32, b[1]) << 16 | @as(u32, b[2]) << 8 | @as(u32, b[3])
                                                                         {

                                .Escribe un u32 como 4 bytes consecutivos big-endian //
                                     } inline fn writeU32(b: []u8, v: u32) void
                                                 ;b[0] = @truncate(v >> 24)
                                                 ;b[1] = @truncate(v >> 16)
                                                  ;b[2] = @truncate(v >> 8)
                                                       ;b[3] = @truncate(v)
                                                                         {

  .Compresión de un bloque de 64 bytes sobre el estado del hash. Sigue §6.2.2 de FIPS 180-4 //
                                    } fn compress(state: *[8]u32, block: [16]u32) void

    Expansión del schedule: 16 palabras → 64. Las nuevas se obtienen .1 //
   combinando cuatro anteriores con dos funciones de mezcla (s0 y s1)    //
        que usan rotación, XOR y desplazamiento. El "+%" es suma con    //
           .truncado u32 (overflow-wrap), tal como exige el estándar    //
                                                ;var w: [64]u32 = undefined
                                               ;for (0..16) |i| w[i] = block[i]
                                                        } |for (16..64) |i
      ;const s0 = rotr(w[i-15], 7) ^ rotr(w[i-15], 18) ^ (w[i-15] >> 3)
       ;const s1 = rotr(w[i-2], 17) ^ rotr(w[i-2], 19) ^ (w[i-2] >> 10)
                                        ;w[i] = w[i-16] +% s0 +% w[i-7] +% s1
                                                                        {

                                .Variables de trabajo: copia del estado actual .2 //
     ;var a = state[0]; var b = state[1]; var c = state[2]; var d = state[3]
     ;var e = state[4]; var f = state[5]; var g = state[6]; var h = state[7]
```

```
                                                .rondas de mezcla no lineal 64 .3 //
                                  .'S1, S0  : combinaciones rotacionales de 'e' y 'a    //
              .ch      : "choose" — multiplexor bit a bit, elige entre f y g según e    //
                               .maj     : "majority" — bit mayoritario entre a, b, c    //
                               .t1 + t2 : se inyecta al top de la cascada cada ronda    //
                                                                  } |for (0..64) |i
                                    ;const S1 = rotr(e, 6) ^ rotr(e, 11) ^ rotr(e, 25)
                                                       ;const ch = (e & f) ^ (~e & g)
                                             ;const t1 = h +% S1 +% ch +% K[i] +% w[i]
                                    ;const S0 = rotr(a, 2) ^ rotr(a, 13) ^ rotr(a, 22)
                                              ;const maj = (a & b) ^ (a & c) ^ (b & c)
                                                                 ;const t2 = S0 +% maj
                                                          ;h = g; g = f; f = e; e = d +% t1
                                                        ;d = c; c = b; b = a; a = t1 +% t2
                                                                                        {

                                     .Acumular las variables de trabajo en el estado .4 //
                           ;state[0] +%= a; state[1] +%= b; state[2] +%= c; state[3] +%= d
                           ;state[4] +%= e; state[5] +%= f; state[6] +%= g; state[7] +%= h
                                                                                          {

        .Hash completo: procesa el mensaje en bloques, padea el último, escribe el resumen //
                                          } pub fn sha256(msg: []const u8, out: *[32]u8) void
                                                                    ;var state = H0
                                                             ;var block: [64]u8 = undefined
                                                         ;var block_w: [16]u32 = undefined

                                        .Procesar bloques completos del mensaje original //
                                                                  ;var i: usize = 0
                                                    } while (i + 64 <= msg.len) : (i += 64)
                                                   ;memcpy(block[0..64], msg[i..i+64])@
                                ;for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4])
                                                             ;compress(&state, block_w)
                                                                                        {

                                       Padding del último bloque: byte 0x80, después ceros, después la //
                      .longitud original (en bits) como u64 big-endian en los 8 últimos bytes //
                                                             ;const remaining = msg.len - i
                                                      ;memcpy(block[0..remaining], msg[i..])@
                                                                  ;block[remaining] = 0x80
                                                 ;const bit_len: u64 = @as(u64, msg.len) * 8

                                                             } if (remaining + 1 + 8 <= 64)
                                                .El padding cabe en el mismo bloque //
                                              ;for (remaining + 1..56) |k| block[k] = 0
                                                                        ;var k: usize = 0
k < 8) : (k += 1) block[56 + k] = @truncate(bit_len >> @as(u6, @intCast((7 - k) * 8)))
                                ;for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4])
                                                             ;compress(&state, block_w)
                                                                                  } else {
                                                .El padding requiere un bloque adicional //
                                                  ;for (remaining + 1..64) |k| block[k] = 0
                                ;for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4])
                                                             ;compress(&state, block_w)
                                                           ;for (0..56) |k| block[k] = 0
                                                                        ;var k: usize = 0
k < 8) : (k += 1) block[56 + k] = @truncate(bit_len >> @as(u6, @intCast((7 - k) * 8)))
                                ;for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4])
                                                             ;compress(&state, block_w)
                                                                                        {

                                             .Escribir el estado final como 32 bytes big-endian //
                                               ;for (0..8) |j| writeU32(out[j*4..j*4+4], state[j])
                                                                                          {
```

```
                                                                .Ejemplo de uso //
                                                              } pub fn main() void
                                                  ;var resumen: [32]u8 = undefined
                                               ;sha256("Cuadernos Lacre", &resumen)
                                  ;for (resumen) |byte| std.debug.print("{x:0>2}", .{byte})
                                                        ;std.debug.print("\n", .{})
                        Imprime: ae6bdea6bbf5476889e0651a31f3dc1612fc61497477e21a95cabae2a6886c3e //
                                                                                {
```

כל כתיבה מחדש בשפה אחרת העוקבת אחר אותו מבנה — קבועים ראשוניים, הרחבת הלו"ז (schedule), שישים וארבעה סבבים, צבירה — מייצרת את אותה תוצאה. לאלגוריתם אין סודות: ערכו טמון בכך שהתכונות המנויות לעיל ממשיכות להתקיים לאחר שני עשורים של קריפטואנליזה ציבורית על ידי אלפי עיניים.

אם תחזור לתחתית המאמר הזה, תראה חותם הקסדצימלי של שישים וארבעה תווים. זהו ה-SHA-256 של הטקסט שזה עתה קראת, בשפה זו. אם היינו מתרגמים את המאמר, החותם היה אחר; אם מילה בגרסה העברית הייתה משתנה, החותם העברי היה משתנה. החותם אינו מגן על התוכן — לשם כך קיימים כלים אחרים — אלא מזהה אותו באופן ייחודי. וזה, צנוע ככל שזה נשמע, מספיק כדי שאף שלב בשרשרת המערכתית לא יוכל לשנות את הנאמר מבלי שזה יורגש. כל השאר — הצפנה, חתימה, זיהוי — נבנה על גבי הרעיון הפשוט הזה.

מקורות וקריאה נוספת

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, אוגוסט 2015. מפרט רשמי של משפחת SHA-2, כולל SHA-256.
- IETF, RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, מאי 2011. גרסה נורמטיבית עבור המממשים.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). פרקים 5 ו-6 מכסים פונקציות hash ואת השימושים הלגיטימיים והבלתי לגיטימיים שלהן.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). דוגמה מעשית לשימוש ב-SHA-256 לשרשור בלוקים במבנה שאינו ניתן לשינוי מעצם בנייתו.
- תקנת (EU) 910/2014 (eIDAS) — מסגרת נותני שירותי חותם זמן מוסמכים. SHA-256 הוא פונקציית הייחוס עבור חתימות וחותמות אלקטרוניות מוסמכות המונפקות באיחוד האירופי.
- מימוש ייחוס ב-Zig: `std.crypto.hash.sha2.Sha256` במאגר הרשמי של השפה (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). זוהי הגרסה האופטימלית והמבוקרת שבה Solo2 אכן משתמשת. מועיל להשוואה עם המימוש הדידקטי של הנספח.

← הקודם CUADERNOS_LIST_SCHREMS_TITLE הבא → CUADERNOS_LIST_KILLSWITCH_TITLE

קריאות אחרונות

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

קחו את המאמר הזה אתכם לכל מקום שתצטרכו.

Markdown ↓ · טקסט פשוט ↓ · PDF ↓

הקובץ יורד למכשיר שלכם. משם תוכלו לשמור אותו, לייבא אותו ל-Solo2 או לשתף אותו היכן שתרצו. Cuadernos לא מחליטה על היעד עבורכם.

חותם שעווה · SHA-256 6839fe4c819a606de3fb7c74496f8a57cf3901165f6f7c12d88e4065126120ea

Cuadernos Lacre · פרסום של Menzuri Gestión S.L. ·
נכתב על ידי R.Eugenio · נערך על ידי צוות Solo2.

אתר זה אינו משתמש בעוגיות (cookies) ואינו טוען משאבים מצד שלישי. הוא משתמש במונה ביקורים אנונימי באירוח עצמי (Umami, בשרת האירופי שלנו) ובמינימום ה-JavaScript הנדרש להעדפת ערכת הנושא הבהירה/כהה שלכם. ללא מעקבים, ללא פרופילינג, ללא שיתוף נתונים. אם תרצו לעקוב אחרינו: RSS.