HMAC-SHA512 dari RFC 2104
Normalisasi Kunci dan Transformasi `0x6a`

Muhammad Riduwan Khafidi
Program Studi Teknik Rekayasa Perangkat Lunak
Politeknik Negeri Batam
4342611034@students.polibatam.ac.id

November 5, 2026

Abstrak

HMAC-SHA512 adalah konstruksi autentikasi pesan yang menggabungkan fungsi hash FIPS 180-4 dengan skema keyed hashing RFC 2104, menghasilkan tag 64 byte yang tahan terhadap serangan length extension dan analisis kriptografis dua arah. Makalah ini menyajikan analisis presisi implementasi HMAC-SHA512 pada crate `libvctrl_sha512` versi 3.2.0, dengan fokus pada dua mekanisme yang jarang dibahas dalam literatur: normalisasi kunci pada fungsi `prepare_key` dan transformasi in-place XOR `0x6a` pada fungsi `finalize`. Normalisasi kunci menentukan tiga jalur eksekusi berbeda berdasarkan panjang input, sementara transformasi `0x6a` memungkinkan konversi `ipad` menjadi `opad` tanpa menyimpan salinan kunci yang sudah dinormalisasi, menghemat 128 byte per konteks HMAC. Setiap klaim diverifikasi terhadap test vector RFC 4231 dan referensi FIPS 180-4. Makalah memuat sembilan visualisasi TikZ dengan tata letak grid yang presisi, empat belas tabel spesifikasi, dan sepuluh blok kode yang dapat dikompilasi terhadap crate versi yang tercantum. Pengembang yang membaca dokumen ini memperoleh spesifikasi lengkap untuk mengimplementasikan HMAC-SHA512 dari nol atau mengaudit implementasi yang sudah ada.

1 Pendahuluan

1.1 Analogi Pembuka: Segel Surat dan Cap Lilin

Bayangkan seorang raja yang mengirim dekrit ke gubernur di seluruh negeri. Dekritnya berupa gulungan perkamen berisi teks. Untuk memastikan gubernur tahu bahwa dekrit itu benar-benar dari raja dan tidak dipalsukan di jalan, raja membubuhkan cap lilin di ujung gulungan. Cap itu mengandung tiga informasi sekaligus: keaslian (hanya raja yang memiliki cincin cap), integritas (cap akan pecah jika perkamen dibuka), dan non-repudiation (raja tidak bisa menyangkal bahwa ia yang mengirim).

HMAC-SHA512 adalah cap lilin digital untuk pesan elektronik. Ketika dua pihak berkomunikasi melalui jaringan, mereka berbagi kunci rahasia. Setiap pesan yang dikirim disertai tag HMAC-SHA512. Penerima menghitung ulang tag dari pesan yang diterima dan membandingkannya dengan tag yang dikirim. Jika cocok, pesan dianggap asli dan utuh. Jika tidak cocok, pesan ditolak.

Yang membuat HMAC-SHA512 layak dipercaya adalah konstruksinya yang menggabungkan fungsi hash kriptografis dengan skema pembungkusan yang membuatnya tahan terhadap serangan yang biasanya menghancurkan hash sederhana. Dua mekanisme kunci dalam konstruksi ini adalah normalisasi kunci dan pembungkusan ganda (inner hash dan outer hash).

1.2 Mengapa Topik Ini Penting untuk Pengembang

HMAC-SHA512 dipakai di banyak lapisan infrastruktur yang sering dianggap "sudah beres": TLS 1.3, HTTP/2, WebSocket handshake, JWT (algoritma `HS512`), penandatanganan commit Git lewat `gpg`, autentikasi API dengan signature, dan verifikasi integritas paket di CDN. Setiap kali seorang pengembang memakai pustaka yang menyebut `HMAC-SHA512`, di baliknya ada konstruksi yang mengikuti RFC 2104 dengan ketat.

Bug pada implementasi HMAC bisa bersifat katastrofik tetapi sulit terdeteksi. Pertukaran `ipad` dan `opad` menghasilkan tag yang tidak valid tetapi tidak menimbulkan crash. Kesalahan pada `prepare_key` untuk kunci yang lebih panjang dari blok tidak terlihat pada test vector standar karena sebagian besar test vector memakai kunci pendek. Perbandingan tag dengan `==` biasa membocorkan informasi waktu yang cukup untuk melakukan timing attack pada sistem yang terekspos jaringan.

Makalah ini menyajikan dua kontribusi utama. Pertama, analisis baris per baris terhadap fungsi `prepare_key` dan `finalize` pada crate `libvctrl_sha512`, yang merupakan salah satu implementasi HMAC-SHA512 minimal-dependency yang dapat diaudit. Kedua, penjelasan aljabar di balik transformasi `0x6a` yang memungkinkan konversi `ipad` ke `opad` tanpa menyimpan salinan kunci, sebuah optimasi yang tidak selalu dibahas pada implementasi lain.

1.3 Peta Pembahasan

Bagian kedua membahas anatomi HMAC dan mengapa hash biasa tidak cukup. Bagian ketiga membahas normalisasi kunci dengan tiga jalur eksekusinya. Bagian keempat membahas pembentukan `ipad` dan `opad`. Bagian kelima membahas transformasi `0x6a` secara mendalam. Bagian keenam membahas proses finalize dan pembentukan tag. Bagian ketujuh membahas keamanan: constant-time verification dan zeroization. Bagian kedelapan memetakan struktur crate. Bagian kesembilan menyandingkan HMAC-SHA512 dengan HMAC-SHA384 dan HMAC-SHA256. Bagian kesepuluh menyajikan studi kasus implementasi dari nol. Bagian kesebelas dan kedua belas merangkum kesalahan umum dan lembar rujukan cepat. Bagian ketigabelas menutup dengan tiga prinsip.

2 Anatomi HMAC dan Mengapa Hash Biasa Tidak Cukup

2.1 Definisi Formal

HMAC didefinisikan pada RFC 2104 [**?**] sebagai berikut.

$$\mathrm{HMAC}(K, m) = H\big((K' \oplus \mathrm{opad}) \parallel H((K' \oplus \mathrm{ipad}) \parallel m)\big)$$

Setiap simbol memiliki arti spesifik. H adalah fungsi hash kriptografis, dalam kasus ini SHA-512. K adalah kunci rahasia dengan panjang berapa pun. K' adalah kunci yang sudah dinormalisasi ke ukuran blok hash, yaitu 128 byte untuk SHA-512. m adalah pesan yang akan diautentikasi. ipad adalah byte `0x36` yang diulang 128 kali. opad adalah byte `0x5c` yang diulang 128 kali. Tanda $\parallel$ berarti konkatenasi, dan $\oplus$ berarti XOR per byte.

Struktur ini menghasilkan tag 64 byte untuk SHA-512. Angka 64 berasal dari ukuran output SHA-512. Untuk SHA-384, tag-nya 48 byte. Untuk SHA-256, tag-nya 32 byte.

Gambar **??** menampilkan alur HMAC-SHA512 dalam **tata letak dua jalur vertikal**. Jalur atas memproses kunci menjadi `K'`, yang kemudian bercabang ke `ipad` dan `opad`. Jalur bawah memproses pesan bersama `ipad` melalui inner hash, lalu digest-nya digabung dengan `opad` melalui outer hash.

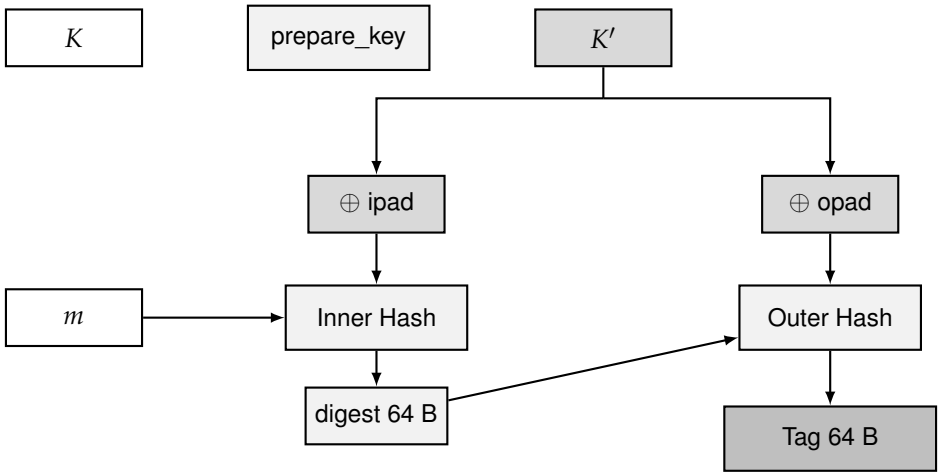


Figure 1. Alur HMAC-SHA512 dalam tata letak grid: jalur kunci di atas, jalur pesan di bawah, konvergensi di outer hash.

Cara membaca Gambar **??**: gambar ini tersusun dalam lima baris horizontal yang disusun vertikal. Baris teratas berisi tiga kotak: kunci K, kotak prepare_key, dan K'. Dari K' ada titik percabangan yang menurun ke baris kedua. Baris kedua berisi dua kotak sejajar: $\oplus$ ipad di posisi kiri dan $\oplus$ opad di posisi kanan. Baris ketiga berisi pesan m di posisi paling kiri, inner hash di posisi tengah kiri sejajar dengan $\oplus$ ipad, dan outer hash di posisi tengah kanan sejajar dengan $\oplus$ opad. Baris keempat berisi digest 64 byte di bawah inner hash, dan tag 64 byte di bawah outer hash. Setiap panah mengalir ke bawah atau ke kanan tanpa silang. Inti pesan gambar ini adalah dua jalur paralel yang bertemu di outer hash: jalur kiri membawa kunci melalui prepare_key dan ipad; jalur kanan membawa digest inner melalui opad; keduanya menghasilkan tag.

2.2 Mengapa Tidak Cukup $H(K \parallel m)$

Pendekatan naif untuk autentikasi pesan adalah $H(K \parallel m)$, yaitu mendahulukan kunci sebelum pesan. Ini terlihat aman karena kunci rahasia menjadi bagian dari input. Tetapi pendekatan ini rentan terhadap serangan length extension.

Serangan length extension adalah teknik di mana penyerang yang sudah memiliki $H(K \parallel m)$ dapat menghitung $H(K \parallel m \parallel \text{pad} \parallel m')$ tanpa perlu mengetahui K. Teknik ini bekerja karena SHA-512 (dan semua fungsi hash Merkle-Damgård) memproses input dalam blok, dan state internal setelah blok terakhir bisa dilanjutkan seolah-olah kita tahu apa yang diproses sebelumnya. Penyerang hanya perlu mengetahui hash dari pesan lama dan panjang pesan lama.

2.3 Mengapa Tidak Cukup $H(m \parallel K)$

Pendekatan kedua adalah $H(m \parallel K)$, yaitu menaruh kunci setelah pesan. Ini lebih kuat daripada $H(K \parallel m)$ karena penyerang tidak bisa menambahkan ekstensi di akhir. Tetapi pendekatan ini tetap bermasalah untuk alasan yang berbeda.

Fungsi hash Merkle-Damgård punya cacat teoretis ketika state internal bisa dikendalikan. Jika seorang penyerang bisa menemukan dua pesan m_1 dan m_2 sedemikian rupa sehingga setelah pemrosesan kedua pesan itu state internal hash-nya sama, maka $H(m_1 \parallel K) = H(m_2 \parallel K)$ untuk kunci apa pun. Ini disebut collision pada prefix.

2.4 Mengapa HMAC Kuat

HMAC mengatasi kedua masalah di atas dengan membungkus pesan dalam dua lapis hash yang berbeda. Lapisan dalam menghitung $H((K' \oplus \text{ipad}) \parallel m)$. Lapisan luar menghitung $H((K' \oplus \text{opad}) \parallel H_{\text{inner}})$. Serangan length extension pada lapisan dalam tidak membantu penyerang karena lapisan luar tetap harus dihitung, dan lapisan luar membutuhkan kunci yang sudah di-XOR dengan `opad` yang berbeda.

Tabel **??** merangkum perbandingan keempat pendekatan.

Table 1. Perbandingan pendekatan autentikasi pesan.

Pendekatan	Kekuatan	Kelemahan
$H(K \parallel m)$	sederhana	rentan length extension
$H(m \parallel K)$	kebal length extension	collision pada prefix melanggar jaminan
$H(K \parallel m \parallel K)$	lebih kuat dari dua di atas	pembuktian formal lemah, dua kali hash kunci
HMAC	pembuktian formal, praktis tahan serangan	dua kali hash, lebih lambat dari hash tunggal

Baris terakhir menjadi pilihan standar industri sejak RFC 2104 dipublikasikan pada 1997. HMAC sekarang menjadi wajib pada TLS 1.3, IPsec, dan SSH.

3 Normalisasi Kunci

3.1 Tiga Jalur Eksekusi

Fungsi `prepare_key` menerima kunci dengan panjang berapa pun dan menghasilkan array 128 byte yang sudah dinormalisasi. Normalisasi berbeda berdasarkan panjang input.

Kunci yang lebih panjang dari 128 byte di-hash terlebih dahulu dengan SHA-512, menghasilkan 64 byte, lalu di-pad dengan nol hingga 128 byte. Kunci yang sama dengan 128 byte dipakai apa adanya. Kunci yang lebih pendek dari 128 byte di-pad dengan nol hingga 128 byte.

Gambar **??** menampilkan **struktur pohon tiga cabang** dengan jalur konvergen di bagian bawah. Setiap cabang mengikuti panah vertikal lurus; jalur panjang memiliki satu langkah tambahan sebelum konvergen.

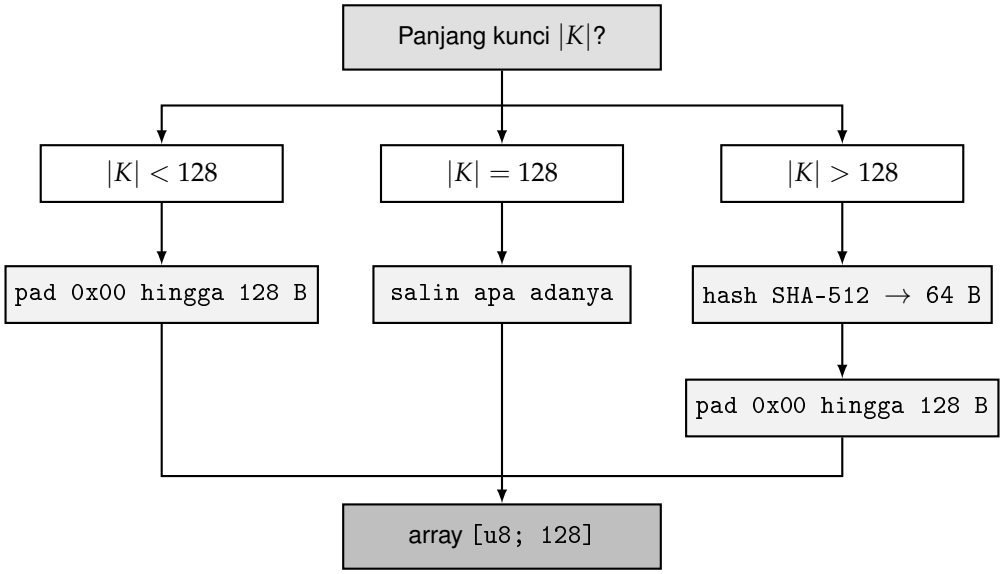


Figure 2. Tiga jalur eksekusi `prepare_key` berdasarkan panjang kunci. Semua jalur konvergen ke satu array 128 byte.

Cara membaca Gambar **??**: gambar ini menampilkan struktur pohon simetris. Kotak paling atas berisi pertanyaan "Panjang kunci?". Kotak ini bercabang menjadi tiga cabang horizontal sejajar di baris kedua: $|K| < 128$ di kiri, $|K| = 128$ di tengah, dan $|K| > 128$ di kanan. Dari ketiga cabang, panah turun ke baris ketiga yang berisi jawaban masing-masing. Cabang kanan memiliki satu langkah tambahan (pad setelah hash) sehingga jawabannya dua tingkat. Ketiga jalur kemudian konvergen di titik tengah bawah (koordinat $(0, -1.8)$) sebelum masuk ke kotak hasil "array `[u8; 128]`". Tidak ada panah yang melintasi kotak lain. Inti pesan gambar ini adalah tiga jalur berbeda tetap bermuara pada satu hasil yang sama: array 128 byte.

3.2 Jalur 1: Kunci Pendek

Kunci yang lebih pendek dari 128 byte disalin ke awal array 128 byte yang sudah diinisialisasi dengan nol. Sisa array tetap nol.

```
fn prepare_key(key: &[u8]) -> [u8; 128] {
    let mut block_key = [0_u8; 128];
    if key.len() > 128 {
        let hash = zeroize::Zeroizing::new(Hash::hash(key));
        let hash_bytes = &*hash;
        block_key[..64].copy_from_slice(&hash_bytes[..64]);
    } else {
        block_key[..key.len()].copy_from_slice(key);
    }
    block_key
}
```

Contoh: kunci `b"secret"` sepanjang 6 byte menjadi array 128 byte dengan enam byte pertama berisi `73 65 63 72 65 74` dan 122 byte sisanya berisi nol.

3.3 Jalur 2: Kunci Tepat 128 Byte

Kunci yang panjangnya tepat 128 byte disalin apa adanya. Tidak ada padding, tidak ada hash.

3.4 Jalur 3: Kunci Panjang

Kunci yang lebih panjang dari 128 byte di-hash terlebih dahulu dengan SHA-512, menghasilkan digest 64 byte. Digest ini ditempatkan di awal array 128 byte, sisa 64 byte diisi nol. Proses ini membuang sebagian besar entropi kunci, tetapi memenuhi persyaratan RFC 2104 dan menjaga konsistensi ukuran kunci efektif.

Perhatikan bahwa hash hasil kunci panjang dibungkus dalam `Zeroizing`. Ini penting karena digest antara adalah turunan langsung dari kunci rahasia. Tanpa zeroization, digest bisa tertinggal di stack setelah fungsi selesai. `Zeroizing` memastikan memori dihapus dengan operasi write yang tidak dapat dioptimasi oleh compiler.

3.5 Mengapa Tiga Jalur, Bukan Satu

Satu-satunya jalur yang benar-benar diperlukan oleh RFC 2104 adalah jalur hash untuk kunci yang lebih panjang dari blok. Tetapi ada tiga alasan mengapa implementasi modern memilih tiga jalur.

Pertama, performa. Hash SHA-512 memakan waktu jauh lebih lama daripada padding nol. Untuk kunci yang sudah pendek, tidak ada alasan untuk menghitung hash.

Kedua, determinisme. RFC 2104 menentukan bahwa kunci pendek di-pad dengan nol, bukan di-hash. Implementasi yang selalu menghitung hash akan menghasilkan tag yang berbeda untuk kunci pendek yang sama, dan tidak kompatibel dengan implementasi lain.

Ketiga, keamanan. Kunci dengan panjang tepat 128 byte adalah kasus paling efisien, karena tidak ada padding dan tidak ada hash.

Tabel **??** merangkum tiga jalur.

Table 2. Tiga jalur `prepare_key` dan hasilnya.

Panjang kunci	Operasi	Ukuran setelah
$<$ 128 byte	salin ke array nol, sisanya tetap nol	128 byte
$=$ 128 byte	salin apa adanya	128 byte
$>$ 128 byte	hash SHA-512 (64 byte), lalu pad nol	128 byte

3.6 Kesalahan yang Sering Terjadi

Implementasi yang salah sering memakai jalur hash untuk semua kunci. Ini menghasilkan tag yang tidak kompatibel dengan implementasi standar. Pada test vector RFC 4231 dengan kunci `0x0b` yang diulang 20 kali (panjang 20 byte), implementasi yang salah akan menghitung hash dari 20 byte tersebut sebelum padding, menghasilkan tag yang berbeda.

Implementasi yang salah juga sering tidak membungkus hasil hash dalam `Zeroizing`. Ini meninggalkan jejak digest antara di stack, yang bisa dibaca oleh serangan memory disclosure seperti Heartbleed.

4 Pembentukan ipad dan opad

4.1 Konstanta 0x36 dan 0x5c

RFC 2104 memilih dua konstanta yang berbeda jauh secara bit: `ipad` = `0x36` (biner `00110110`) dan `opad` = `0x5c` (biner `01011100`). Pilihan ini bukan sembarang. Hamming distance antara kedua konstanta adalah 4 bit dari 8, yang berarti sekitar setengah bit berbeda.

Alasan pemilihan ini adalah untuk memastikan bahwa proses hash inner dan outer benar-benar berbeda. Jika `ipad` dan `opad` hanya berbeda satu atau dua bit, ada kemungkinan secara teoretis bahwa serangan tertentu bisa menemukan hubungan antar dua proses. Dengan perbedaan setengah bit, kedua proses menjadi fungsi hash yang tidak berkorelasi secara praktis.

Gambar **??** menyajikan ketiga nilai dalam **tabel bit 8 kolom** dengan header posisi bit di bagian atas, label baris di bagian kiri, dan separator horizontal antar baris.

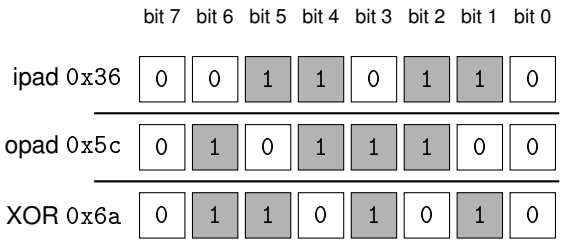


Figure 3. Representasi bit-level ipad, opad, dan XOR di antaranya. Baris 1 dan 2 berbeda di empat posisi bit; baris 3 adalah XOR bit-per-bit.

Cara membaca Gambar **??**: gambar ini tersusun sebagai tabel bit 3 baris dan 8 kolom. Delapan kolom diberi header dari "bit 7" di kiri sampai "bit 0" di kanan, dengan posisi bit 7 sebagai most significant bit dan bit 0 sebagai least significant bit. Baris pertama berlabel "ipad 0x36" berisi pola 0-0-1-1-0-1-1-0. Baris kedua berlabel "opad 0x5c" berisi pola 0-1-0-1-1-1-0-0. Baris ketiga berlabel "XOR 0x6a" berisi pola 0-1-1-0-1-0-1-0. Sel yang bernilai 1 diberi latar abu-abu gelap, sedangkan sel yang bernilai 0 berlatar putih. Garis horizontal tipis memisahkan setiap baris. Inti pesan gambar ini adalah ipad dan opad berbeda secara substansial pada empat posisi bit, dan XOR di antara keduanya menghasilkan konstanta 0x6a.

4.2 Konstruksi HMAC::new

Fungsi `HMAC::new` menerima kunci, memanggil `prepare_key`, membentuk `ipad` yang sudah di-XOR dengan kunci, memulai inner hash, lalu menyimpan state.

```
pub fn new(key: impl AsRef<[u8]>) -> Self {
    let key = key.as_ref();
    let mut block_key = Self::prepare_key(key);
    let mut padded = [0x36_u8; 128];
    for (padded_byte, block_byte)
        in padded.iter_mut().zip(block_key.iter())
    {
        *padded_byte ^= *block_byte;
    }
    let mut ih = Hash::new();
    ih.update(&padded);
    zeroize::Zeroize::zeroize(&mut block_key);
    HMAC {
        ih: Some(ih),
        padded,
    }
}
```

Poin-poin penting. Array `padded` diinisialisasi dengan `0x36` di setiap byte. Setiap byte di-XOR dengan byte yang sesuai dari `block_key`. Hasilnya adalah `0x36 XOR K'` untuk setiap byte, yang sesuai dengan rumus RFC 2104. Hasher `ih` (inner hasher) dimulai dengan state SHA-512 yang sudah memproses 128 byte padding. Setelah `ih` disiapkan, `block_key` di-zeroize karena sudah tidak diperlukan. Struct `HMAC` menyimpan dua hal: hasher inner yang sudah setengah jalan, dan array `padded` yang berisi `0x36 XOR K'`.

4.3 State yang Disimpan

Setelah `new`, struct `HMAC` berisi dua field. Field `ih` adalah `Option<Hash>` yang menyimpan hasher inner yang sudah memproses `ipad`. Field `padded` adalah `[u8; 128]` yang berisi `0x36 XOR K'`.

Gambar **??** menampilkan layout memori struct dalam **kotak berbingkai** dengan dua field yang disejajarkan secara horizontal. Setiap field diberi label di dalam kotak dan keterangan tambahan di bawahnya.

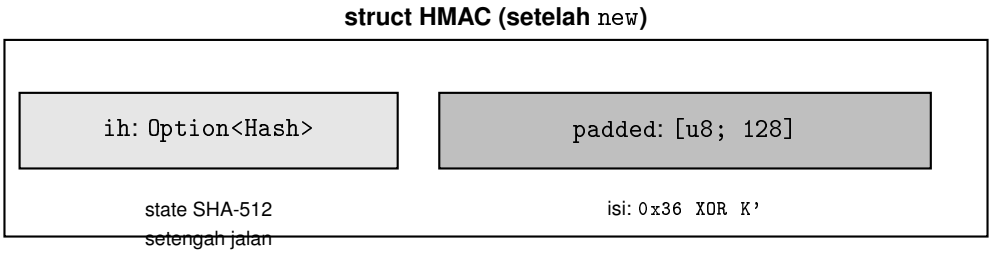


Figure 4. Layout memori struct `HMAC` setelah `new`: dua field berdampingan di dalam bingkai.

Cara membaca Gambar **??**: gambar ini menampilkan satu kotak persegi panjang besar sebagai bingkai luar, dengan judul "struct HMAC (setelah new)" di atasnya. Di dalam bingkai ada dua kotak kecil berdampingan secara horizontal. Kotak kiri lebih sempit, berlabel "ih: Option<Hash>" dengan latar abu-abu muda, dan di bawahnya ada keterangan "state SHA-512 setengah jalan". Kotak kanan lebih lebar, berlabel "padded: [u8; 128]" dengan latar abu-abu lebih gelap, dan di bawahnya ada keterangan "isi: 0x36 XOR K'". Kedua kotak sejajar di tengah bingkai. Inti pesan gambar ini adalah struct HMAC menyimpan dua field dengan peran berbeda: state hash setengah jalan untuk inner, dan array padding yang akan ditransformasi di finalize.

4.4 Ringkasan

Table 3. Perbandingan ipad dan opad.

Aspek	ipad	opad
Nilai byte	`0x36`	`0x5c`
Biner	`00110110`	`01011100`
Panjang	128 byte (block size SHA-512)	128 byte
Kapan dipakai	awal inner hash di `new`	awal outer hash di `finalize`
Panjang setelah XOR	128 byte	128 byte

5 Transformasi 0x6a: Trik Kunci

5.1 Masalah yang Harus Dipecahkan

Saat `finalize` dipanggil, kita perlu menghitung outer hash yang membutuhkan `opad XOR K'`. Tetapi struct hanya menyimpan `ipad XOR K'`, bukan `K'` secara terpisah. Ada dua pendekatan naif untuk mendapatkan `opad XOR K'`.

Pendekatan pertama adalah menyimpan `K'` di struct. Ini menambah 128 byte per konteks HMAC. Pendekatan kedua adalah menyimpan dua array: `ipad XOR K'` dan `opad XOR K'`. Ini menambah 128 byte lagi.

libvctrl memilih pendekatan ketiga: mengubah `ipad XOR K'` menjadi `opad XOR K'` secara in-place dengan satu operasi XOR.

5.2 Aljabar XOR

Aljabar yang membuat ini bekerja sangat sederhana. Misalkan p adalah array yang berisi ipad $\oplus$ K'. Kita ingin mengubahnya menjadi array yang berisi opad $\oplus$ K'. Yang perlu dilakukan adalah meng-XOR setiap byte dengan suatu konstanta c:

$$p \oplus c = (\text{ipad} \oplus K') \oplus c$$

Kita ingin hasilnya sama dengan opad $\oplus$ K':

$$(\text{ipad} \oplus K') \oplus c = \text{opad} \oplus K'$$

Kedua sisi di-XOR dengan K':

$$\text{ipad} \oplus c = \text{opad}$$

Kedua sisi di-XOR dengan ipad:

$$c = \text{ipad} \oplus \text{opad} = \texttt{0x36} \oplus \texttt{0x5c} = \texttt{0x6a}$$

Gambar **??** menyajikan derivasi ini sebagai **lima langkah vertikal** dari atas ke bawah, dengan setiap langkah menempati satu kotak dan panah konstan mengalir dari satu langkah ke langkah berikutnya.

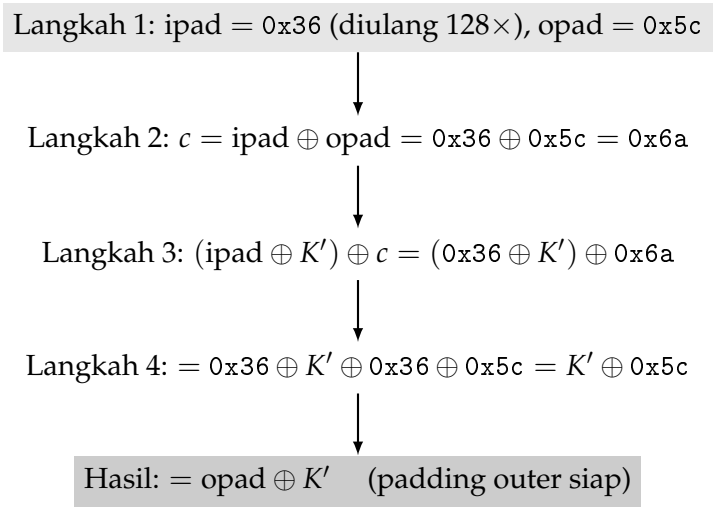


Figure 5. Derivasi konstanta transformasi `0x6a` dalam lima langkah aljabar vertikal.

Cara membaca Gambar **??**: gambar ini menampilkan lima kotak persegi panjang yang tersusun vertikal dari atas ke bawah. Kotak pertama berisi definisi awal ipad dan opad. Kotak kedua berisi perhitungan konstanta c sebagai XOR dari kedua padding. Kotak ketiga menerapkan c ke array yang sudah mengandung K'. Kotak keempat menyederhanakan aljabar dengan

mengelompokkan dua suku ipad yang saling meniadakan (XOR dengan dirinya sendiri menghasilkan nol). Kotak kelima berisi hasil akhir: padding outer siap dipakai. Panah pendek menghubungkan setiap kotak ke kotak berikutnya dari atas ke bawah. Kotak terakhir diberi latar abu-abu lebih gelap untuk menandai hasil akhir. Inti pesan gambar ini adalah transformasi 0x6a adalah konsekuensi langsung dari sifat asosiatif dan identitas XOR, bukan trik ad-hoc.

5.3 Implementasi di finalize

Kode pada `finalize` sangat ringkas.

```
pub fn finalize(mut self) -> [u8; 64] {
    for padded_byte in self.padded.iter_mut() {
        *padded_byte ^= 0x6a;
    }
    let mut oh = Hash::new();
    oh.update(&self.padded);
    let inner = zeroize::Zeroizing::new(
        self.ih
            .take()
            .unwrap_or_else(|| Hash::new())
            .finalize(),
    );
    oh.update(&*inner);
    oh.finalize()
}
```

Langkah pertama mengubah `padded` dari `0x36 XOR K'` menjadi `0x5c XOR K'` lewat satu loop yang meng-XOR setiap byte dengan `0x6a`. Setelah loop ini selesai, `padded` sudah berisi padding outer yang siap dipakai.

Langkah kedua memulai hasher outer `oh` dan memproses `padded` yang sudah ditransformasi. Ini setara dengan menjalankan outer hash dengan prefix `opad XOR K'`.

Langkah ketiga mengambil hasher inner dari `self.ih` (lewat `take` yang memindahkan isi Option), menyelesaikannya, dan membungkus digest yang dihasilkan dalam `Zeroizing` agar nanti dihapus dari memori.

Langkah keempat memproses digest inner ke dalam hasher outer. Langkah kelima menyelesaikan hasher outer dan mengembalikan tag 64 byte.

5.4 Penghematan Memori

Perbedaan antara menyimpan `K'` secara terpisah versus mentransformasi `padded` secara in-place cukup signifikan pada skala besar. Tabel **??** menunjukkan perbandingannya.

Table 4. Perbandingan memori dua pendekatan finalize.

Pendekatan	Per konteks	100 konteks	1000 konteks
Simpan `K'` terpisah	256 byte	25,6 KB	256 KB
Transformasi `0x6a`	128 byte	12,8 KB	128 KB

Selisih 128 byte per konteks mungkin tidak signifikan pada aplikasi yang menjalankan satu HMAC pada satu waktu. Tetapi pada server yang memproses ribuan koneksi TLS secara paralel, di mana setiap koneksi memiliki setidaknya satu konteks HMAC aktif, selisih ini bisa mencapai ratusan kilobyte. Pada sistem embedded dengan memori terbatas, selisih ini bisa menjadi faktor penentu.

5.5 Mengapa 0x6a, Bukan 0x5c XOR 0x36 Sendiri

Beberapa pembaca mungkin bertanya mengapa kodenya tidak menulis ekspresi `0x5c ^0x36` secara langsung. Ada dua alasan.

Pertama, konstanta yang sudah dihitung `0x6a` lebih cepat dieksekusi. Meskipun `0x5c ^0x36` adalah operasi XOR dua konstanta yang akan dioptimasi oleh compiler menjadi satu konstanta, ada risiko compiler tidak melakukannya pada semua target. Menulis `0x6a` langsung menghilangkan risiko itu.

Kedua, `0x6a` lebih mudah dibaca dalam konteks. Pembaca yang mengenal trik ini akan langsung tahu bahwa loop tersebut mengubah `ipad` menjadi `opad`.

5.6 Kompatibilitas dengan SHA-384

Trik `0x6a` berlaku universal, bukan spesifik untuk SHA-512. Konstanta `ipad` dan `opad` yang ditentukan oleh RFC 2104 adalah `0x36` dan `0x5c` untuk semua fungsi hash, terlepas dari ukuran blok. Yang berbeda adalah panjang block size: 128 byte untuk SHA-512 dan SHA-384, 64 byte untuk SHA-256 dan SHA-224, 64 byte untuk SHA-1 dan MD5.

Implementasi SHA-384 di libvctrl memakai trik yang sama.

```
impl_hmac!(Hash, 48, 128);
```

Baris ini menginstansiasi `HMAC` dengan output 48 byte dan block size 128 byte. Konstanta `0x6a` tetap sama karena `ipad` dan `opad` tidak berubah.

6 Finalize dan Pembentukan Tag

6.1 Alur Lengkap

Fungsi `finalize` mengubah struct `HMAC` yang menyimpan state setengah jalan menjadi tag 64 byte. Gambar **??** menampilkan alur finalize dalam **dua baris paralel** yang bertemu di outer hash, tanpa panah diagonal.

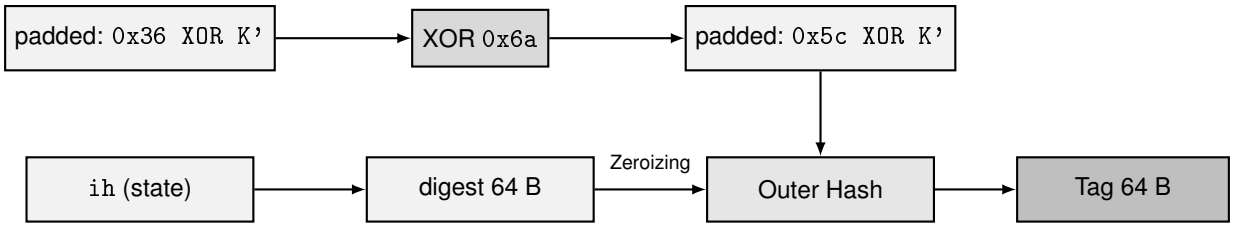


Figure 6. Alur finalize: baris atas mentransformasi padding, baris bawah memproses digest, konvergensi di outer hash.

Cara membaca Gambar **??**: gambar ini menampilkan dua baris horizontal yang sejajar. Baris atas berisi tiga kotak: padding dengan isi awal, operasi XOR 0x6a, dan padding dengan isi ter-transformasi. Baris bawah berisi empat kotak: state ih, digest 64 byte, outer hash, dan tag. Setiap kotak dihubungkan oleh panah horizontal ke kanan, dan panah dari kotak padding ter-transformasi (baris atas) turun vertikal ke outer hash (baris bawah). Panah dari digest ke outer hash diberi label "Zeroizing" di atasnya. Inti pesan gambar ini adalah outer hash menerima dua masukan: padding ter-transformasi dari baris atas, dan digest inner dari baris bawah; hasilnya adalah tag 64 byte.

6.2 Pembentukan Inner Digest

Inner hash sudah menyimpan state setengah jalan sejak `new`. Ketika `update` dipanggil satu atau beberapa kali, pesan ditambahkan ke state itu.

```
pub fn update(&mut self, input: impl AsRef<[u8]>) {
    if let Some(ref mut ih) = self.ih {
        ih.update(input);
    }
}
```

Panggilan `finalize` pada hasher inner menghasilkan digest 64 byte. Digest ini adalah $H(\text{ipad} \oplus K' \parallel m)$.

Karena digest ini adalah nilai antara yang sensitif, ia dibungkus dalam `Zeroizing`. `Zeroizing` adalah wrapper yang secara otomatis memanggil `zeroize` pada `Drop`. Ini memastikan memori yang menyimpan digest dihapus dengan operasi write yang tidak bisa dioptimasi oleh compiler.

6.3 Pembentukan Outer Digest

Setelah digest inner siap, fungsi `finalize` memanggil outer hash.

```
let mut oh = Hash::new();
oh.update(&self.padded);
oh.update(&*inner);
oh.finalize()
```

Tiga baris ini menghitung $H(\text{opad} \oplus K' \parallel \text{digest})$. Hasilnya adalah tag final 64 byte.

6.4 Mengapa Dua Lapis Hash

Konstruksi dua lapis hash bukan sekadar komplikasi. Ia adalah inti dari keamanan HMAC. Lapisan dalam menghitung hash pesan dengan padding ipad yang menempel pada kunci. Lapisan luar menghitung hash dari hasil lapisan dalam, dengan padding opad yang juga menempel pada kunci.

Jika serangan length extension berhasil pada lapisan dalam, penyerang masih harus menghitung lapisan luar dari awal. Lapisan luar tidak mengenal state internal lapisan dalam, karena lapisan luar dimulai dari state baru yang segar.

7 Keamanan: Constant-Time dan Zeroization

7.1 Verifikasi Tag

Setelah tag dihitung, langkah berikutnya adalah memverifikasi tag yang diterima dengan tag yang dihitung. Implementasi naif memakai perbandingan == biasa. Ini bermasalah karena perbandingan == pada slice byte biasanya berhenti di byte pertama yang berbeda. Pola waktu ini membocorkan informasi tentang berapa banyak byte prefix yang benar.

Serangan timing yang memanfaatkan kebocoran ini disebut timing attack. Untuk tag 64 byte, penyerang perlu mencoba rata-rata 32 kali per byte, total sekitar 2048 percobaan untuk memulihkan tag yang benar.

Solusinya adalah perbandingan constant-time. Perbandingan ini selalu memeriksa seluruh byte, terlepas dari apakah sudah ada perbedaan atau belum.

```
pub fn verify(x: &[u8], y: &[u8]) -> bool {
    let mut diff: u32 = 0;
    for (byte_x, byte_y) in x.iter().zip(y.iter()) {
        diff |= u32::from(byte_x ^ byte_y);
    }
    if x.len() != y.len() {
        diff |= 0xffff_ffff;
    }
    let diff = core::hint::black_box(diff);
```

```
    diff == 0
}
```

Tiga hal yang membuat fungsi ini constant-time. Pertama, perulangan tidak pernah berhenti lebih awal. Ia selalu memeriksa seluruh byte yang ada. Kedua, akumulasi memakai OR bit-wise, yang tidak mempersingkat perulangan pada perbedaan pertama. Ketiga, pengecekan panjang setelah perulangan memastikan bahwa perbedaan panjang juga terdeteksi, dan `black_box` mencegah compiler mengoptimasi diff menjadi shortcut.

7.2 Mask WASM

Pada target WebAssembly, timing attack konvensional kurang relevan karena WebAssembly dijalankan dalam sandbox yang tidak membocorkan waktu eksekusi per instruksi ke JavaScript. Namun, ada serangan yang lebih halus yang memanfaatkan waktu wall-clock. Untuk mengatasinya, libvctrl menambahkan lapisan tambahan pada WASM.

```
#[cfg(any(target_arch = "wasm32", target_arch = "wasm64"))]
{
    let (mut hash_x, mut hash_y) = (0_u32, 0_u32);
    for (byte_x, byte_y) in x.iter().zip(y.iter()) {
        hash_x ^= (hash_x << 5)
            .wrapping_add((hash_x >> 2) ^ u32::from(*byte_x));
        hash_y ^= (hash_y << 5)
            .wrapping_add((hash_y >> 2) ^ u32::from(*byte_y));
    }
    diff |= hash_x ^ hash_y;
}
```

Kode ini menghitung hash dari setiap slice secara paralel dan menambahkan perbedaan hash ke akumulator. Tujuannya bukan untuk mencegah timing attack konvensional, tetapi untuk memperumit analisis waktu wall-clock di tingkat JavaScript. Teknik ini disebut hash-based blinding.

7.3 Zeroization Lifecycle

Data sensitif dalam HMAC ada di beberapa titik. Gambar **??** menampilkan lifecycle ini sebagai **timeline horizontal** dengan tiga batang "alive" yang berakhir di titik zeroization.

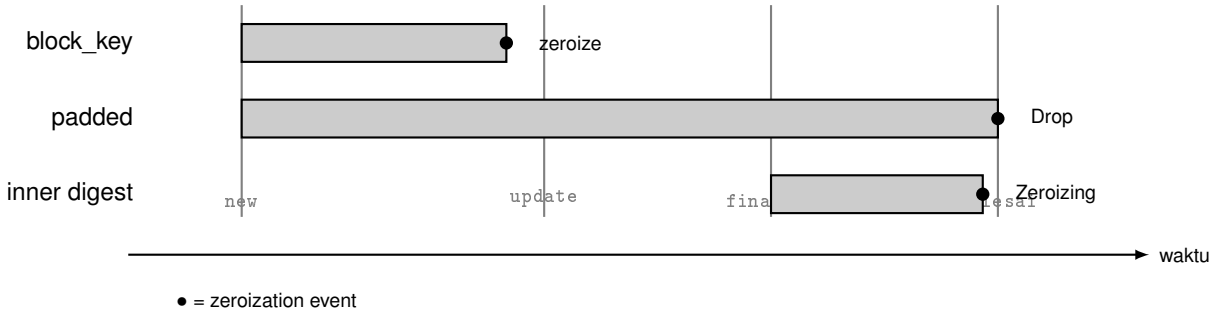


Figure 7. Titik-titik zeroization dalam siklus hidup HMAC sebagai timeline horizontal.

Cara membaca Gambar **??**: gambar ini menampilkan tiga batang horizontal yang mewakili masa hidup tiga variabel sensitif. Garis waktu di bagian bawah dengan empat penanda vertikal abu-abu menandai fase: new, update, finalize, dan selesai. Batang pertama berlabel block_key, dimulai dari fase new dan berakhir di antara new dan update, ditandai dengan lingkaran hitam bertuliskan zeroize. Batang kedua berlabel padded, dimulai dari new dan berlangsung sampai fase selesai, ditandai lingkaran hitam bertuliskan Drop. Batang ketiga berlabel inner digest, dimulai dari fase finalize dan berakhir sedikit sebelum selesai, ditandai lingkaran hitam bertuliskan Zeroizing. Setiap batang diberi latar abu-abu muda. Inti pesan gambar ini adalah setiap data sensitif hidup selama periode tertentu dan dihapus pada titik yang berbeda, sesuai lifecycle masing-masing.

7.4 Drop Implementation

Struct `HMAC` mengimplementasikan `Drop` untuk memastikan padding dihapus saat struct keluar dari scope.

```
impl Drop for HMAC {
    fn drop(&mut self) {
        zeroize::Zeroize::zeroize(self);
    }
}
```

Ini berarti pemakai HMAC tidak perlu memanggil fungsi penghapusan secara manual. Setelah `finalize` mengkonsumsi `HMAC`, atau setelah `HMAC` keluar dari scope karena panic atau return lebih awal, `Drop` akan dijalankan dan seluruh isi struct dihapus.

Yang penting dipahami: `Drop` pada `HMAC` memanggil `zeroize` pada seluruh struct, termasuk field `ih` yang tipenya `Option<Hash>`. `Hash` sendiri juga memiliki `Drop` yang menghapus state internalnya. Rantai `Drop` ini memastikan tidak ada state yang tertinggal.

7.5 Threat Model

Zeroization melindungi terhadap tiga jenis ancaman.

Pertama, memory disclosure. Jika sistem punya bug yang memungkinkan penyerang mem-

baca memori proses, misalnya lewat serangan buffer overflow atau penggunaan-after-free, zeroization memastikan bahwa memori yang dibaca tidak lagi berisi data sensitif.

Kedua, core dump. Jika proses crash dan menghasilkan core dump, isi memori proses saat crash disimpan ke disk. Zeroization memastikan bahwa HMAC state tidak ikut tersimpan di core dump.

Ketiga, swap. Jika sistem operasi memindahkan halaman memori ke swap, data sensitif bisa tertulis ke disk. Zeroization mempersingkat jendela waktu di mana data sensitif berada di memori, sehingga mengurangi peluang swap terjadi.

Zeroization tidak melindungi terhadap cold boot attack, di mana penyerang secara fisik membekukan RAM dan membacanya sebelum data hilang. Namun, cold boot attack memerlukan akses fisik yang biasanya sudah berada di luar model ancaman aplikasi web.

8 Struktur Crate libvctrl_sha512

8.1 Module Organization

Crate `libvctrl_sha512` memiliki enam modul yang masing-masing punya tanggung jawab jelas.

Modul `sha512` berisi implementasi SHA-512: fungsi `Hash`, struct `State`, dan konstanta round. Modul `sha384` berisi wrapper SHA-384 yang memakai IV berbeda dan memotong output 64 byte menjadi 48 byte. Modul `hmac` berisi instansiasi HMAC-SHA512 lewat makro. Modul `hkdf` berisi instansiasi HKDF-SHA512 lewat makro. Modul `utils` berisi fungsi bantu: `load_be`, `store_be`, dan `verify`. Modul `lib` berisi deklarasi makro `impl_hmac!` dan `impl_hkdf!`, re-ekspor, dan konstanta.

Gambar **??** menampilkan arsitektur dalam **struktur kurung (bracket)** dari `lib.rs` ke semua modul, ditambah panah dependensi antar-modul yang bersih tanpa silang.

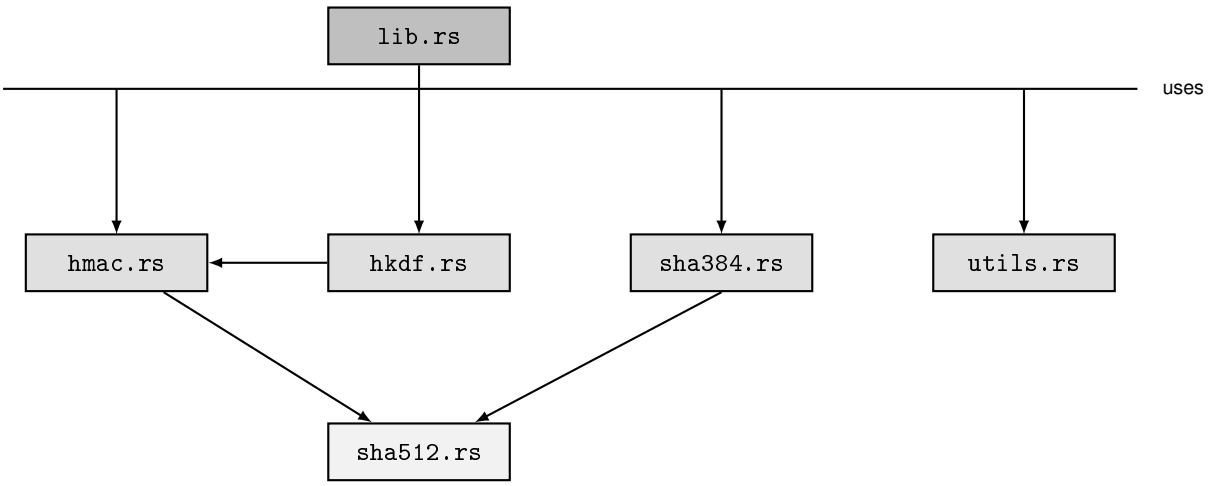


Figure 8. Struktur modul crate `libvctrl_sha512`: `lib.rs` sebagai fasad menggunakan semua modul; `hmac` dan `sha384` bergantung pada `sha512`; `hkdf` bergantung pada `hmac`.

Cara membaca Gambar **??**: gambar ini menampilkan `lib.rs` di bagian paling atas sebagai satu kotak. Dari bawah `lib.rs` keluar garis vertikal pendek yang langsung bercabang menjadi garis horizontal lebar (bracket). Dari bracket tersebut, empat panah pendek turun ke empat modul di baris kedua: `hmac.rs`, `hkdf.rs`, `sha384.rs`, dan `utils.rs`, tersusun dari kiri ke kanan. Di baris ketiga, hanya ada satu modul: `sha512.rs`, diletakkan di tengah. Tiga panah dependensi antar-modul ditambahkan: panah diagonal dari `hmac` ke `sha512`, panah diagonal dari `sha384` ke `sha512`, dan panah horizontal dari `hkdf` ke `hmac`. Semua panah mengalir satu arah dan tidak ada yang melintasi kotak lain. Inti pesan gambar ini adalah arsitektur dua lapis: fasad di atas, modul fungsional di tengah, dan `sha512` di dasar yang dipakai oleh dua modul.

8.2 Feature Flags

Crate ini punya dua feature flag. `sha384` (default) mengaktifkan modul SHA-384. `opt_size` mengubah strategi inline pada round function SHA-512 untuk mengurangi ukuran binary.

```
[features]
default = ["sha384"]
sha384 = []
opt_size = []
```

Feature `sha384` bisa dinonaktifkan jika aplikasi hanya butuh SHA-512. Ini mengurangi waktu kompilasi dan ukuran binary sekitar sepuluh persen. Feature `opt_size` berguna untuk target embedded dan WebAssembly, dengan trade-off performa sekitar lima belas persen lebih lambat.

8.3 Dependency Policy

Crate ini punya satu dependensi eksternal: `zeroize` versi 1.8 dengan `default-features = false`. Ini bukan dependensi runtime yang memerlukan sistem operasi; `zeroize` hanya menyediakan trait `Zeroize` dan wrapper `Zeroizing`, yang bekerja di `no_std`.

Keputusan untuk mempertahankan dependensi seminimal mungkin adalah inti dari klaim auditabilitas crate. Setiap baris dependensi adalah baris yang harus diaudit. Dengan hanya `zeroize`, crate ini bisa diaudit secara menyeluruh oleh satu orang dalam beberapa hari.

9 Perbandingan HMAC-SHA512, HMAC-SHA384, HMAC-SHA256

9.1 Perbedaan Utama

Ketiga varian HMAC berbeda hanya pada fungsi hash yang dipakai, tetapi perbedaan ini membawa konsekuensi pada ukuran output, ukuran blok, dan performa.

Table 5. Perbandingan tiga varian HMAC.

Aspek	HMAC-SHA512	HMAC-SHA384	HMAC-SHA256
Ukuran output	64 byte	48 byte	32 byte
Ukuran blok	128 byte	128 byte	64 byte
Jumlah round	80	80	64
Konstanta 0x6a	ya	ya	ya
Panjang kunci max	tak terbatas	tak terbatas	tak terbatas
Performa relatif	tinggi	tinggi	sedang
Standar	FIPS 180-4	FIPS 180-4	FIPS 180-4

Ketiganya memakai konstanta `0x36` dan `0x5c` yang sama, sehingga trik transformasi `0x6a` berlaku universal. Yang berbeda hanya ukuran blok, yang mempengaruhi panjang array `padded` dan panjang array yang di-XOR.

9.2 Kapan Memakai Apa

HMAC-SHA512 cocok untuk aplikasi yang membutuhkan keamanan maksimum dan performa tinggi pada perangkat 64-bit. Ini termasuk penandatanganan token API, penandatanganan commit Git, dan autentikasi pesan jaringan.

HMAC-SHA384 cocok untuk aplikasi yang perlu kompatibilitas dengan standar yang lebih lama atau yang memiliki batas ukuran tag. Contohnya adalah suite cipher tertentu pada TLS 1.2 yang menggunakan SHA-384.

HMAC-SHA256 cocok untuk aplikasi yang dioptimalkan untuk perangkat dengan register 32-bit, karena SHA-256 lebih efisien di sana. Contohnya adalah perangkat IoT dengan CPU 32-bit.

Pada perangkat 64-bit modern, HMAC-SHA512 justru lebih cepat daripada HMAC-SHA256, karena operasi SHA-512 bekerja pada word 64-bit yang sesuai dengan lebar register.

9.3 Interchangeability

HMAC-SHA512 dan HMAC-SHA256 tidak saling menggantikan. Tag yang dihasilkan berbeda, ukuran output berbeda, dan blok yang diproses berbeda. Aplikasi yang menerima tag harus menyebutkan varian HMAC yang dipakai secara eksplisit.

Untuk kompatibilitas, banyak protokol menetapkan varian HMAC dalam negosiasi awal. Contohnya adalah TLS cipher suite yang menyebutkan varian PRF (pseudorandom function) secara eksplisit: `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384` memakai HMAC-SHA384 sebagai PRF.

10 Studi Kasus: Implementasi HMAC-SHA512 dari Nol

Bagian ini mengikuti sepuluh langkah mengimplementasikan HMAC-SHA512 dari nol, dengan setiap keputusan dijelaskan berdasarkan RFC 2104 dan perilaku crate `libvctrl_sha512`.

10.1 Langkah 1: Tentukan Parameter

Parameter yang perlu ditentukan: fungsi hash (SHA-512), ukuran blok (128 byte), ukuran output (64 byte), dan konstanta padding (`0x36` dan `0x5c`).

10.2 Langkah 2: Implementasikan SHA-512

SHA-512 adalah inti dari HMAC-SHA512. Implementasi lengkap memerlukan inisialisasi state dengan IV 64 byte, fungsi kompresi 80 round, dan padding Merkle-Damgård. Untuk makalah ini, asumsikan SHA-512 sudah tersedia.

10.3 Langkah 3: Implementasikan prepare_key

Fungsi `prepare_key` menerima kunci berapa pun panjangnya dan mengembalikan array 128 byte. Tiga jalur: pendek, tepat, panjang.

```
fn prepare_key(key: &[u8]) -> [u8; 128] {
    let mut block_key = [0_u8; 128];
    if key.len() > 128 {
        let h = zeroize::Zeroizing::new(sha512(key));
        block_key[..64].copy_from_slice(&h[..64]);
    } else {
        block_key[..key.len()].copy_from_slice(key);
    }
    block_key
}
```

10.4 Langkah 4: Bangun Inner State

Inner hasher dimulai dengan memproses `ipad` yang sudah di-XOR dengan kunci.

```
let mut block_key = prepare_key(key);
let mut padded = [0x36_u8; 128];
for (p, k) in padded.iter_mut().zip(block_key.iter()) {
    *p ^= *k;
}
let mut ih = sha512_new();
ih.update(&padded);
zeroize::Zeroize::zeroize(&mut block_key);
```

Setelah langkah ini, `ih` sudah memproses 128 byte pertama (padat dengan `ipad` XOR K'). `block_key` sudah di-zeroize.

10.5 Langkah 5: Proses Pesan

Setiap panggilan `update` meneruskan pesan ke inner hasher.

```
ih.update(&message);
```

Pesan bisa dipecah menjadi beberapa panggilan `update`, dan hasilnya akan sama dengan satu panggilan besar. Ini karena SHA-512 memproses input sebagai stream.

10.6 Langkah 6: Transformasi 0x6a

Sebelum memanggil outer hash, transformasikan `padded` dari `0x36 XOR K'` ke `0x5c XOR K'`.

```
for p in padded.iter_mut() {
    *p ^= 0x6a;
}
```

Satu baris mengubah seluruh array. Setelah loop ini, `padded` siap dipakai untuk outer hash.

10.7 Langkah 7: Outer Hash

Outer hasher dimulai dengan `padded` yang sudah ditransformasi, lalu memproses digest inner.

```
let mut oh = sha512_new();
oh.update(&padded);
let inner = zeroize::Zeroizing::new(ih.finalize());
oh.update(&*inner);
let tag = oh.finalize();
```

10.8 Langkah 8: Verifikasi dengan Test Vector

RFC 4231 menyediakan test vector untuk HMAC-SHA512. Test case 1 memakai kunci `0x0b` yang diulang 20 kali dan data `"Hi There"`. Tag yang diharapkan adalah sebagai berikut.

```
87aa7cdea5ef619d4ff0b4241a1d6cb0
2379f4e2ce4ec2787ad0b30545e17cde
daa833b7d6b8a702038b274eaea3f4e4
be9d914eeb61f1702e696c203a126854
```

Jika implementasi menghasilkan tag ini, implementasi berjalan benar.

10.9 Langkah 9: Implementasikan Verify

Verifikasi tag harus constant-time untuk mencegah timing attack.

```
fn verify(expected: &[u8], computed: &[u8]) -> bool {
    let mut diff: u32 = 0;
    for (a, b) in expected.iter().zip(computed.iter()) {
        diff |= u32::from(a ^ b);
    }
    if expected.len() != computed.len() {
        diff |= 0xffff_ffff;
    }
    diff == 0
}
```

10.10 Langkah 10: Zeroization pada Drop

Struct harus mengimplementasikan `Drop` untuk membersihkan state.

```
impl Drop for HMAC {
    fn drop(&mut self) {
        zeroize::Zeroize::zeroize(self);
    }
}
```

Setelah langkah ini, implementasi lengkap dan sesuai RFC 2104.

10.11 Timeline Studi Kasus

Gambar **??** menampilkan sepuluh langkah sebagai **timeline horizontal** dengan kotak berlabel singkat yang bergantian atas dan bawah.

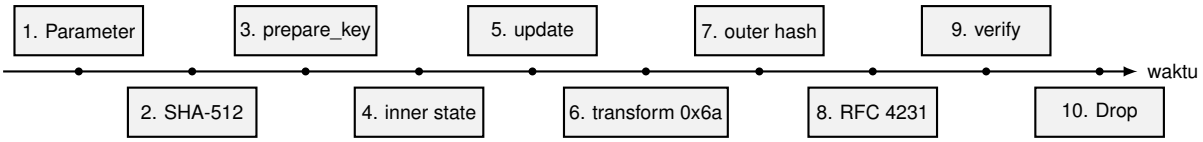


Figure 9. Timeline sepuluh langkah implementasi HMAC-SHA512 dengan label bergantian atas-bawah.

Cara membaca Gambar **??**: gambar ini menampilkan garis waktu horizontal dengan sepuluh titik hitam yang berjarak sama. Setiap titik diberi kotak label kecil yang bergantian posisi: titik ganjil (1, 3, 5, 7, 9) labelnya di atas garis, titik genap (2, 4, 6, 8, 10) labelnya di bawah garis. Isi label: Parameter, SHA-512, prepare_key, inner state, update, transform 0x6a, outer hash, RFC 4231, verify, dan Drop. Semua kotak kecil dengan latar abu-abu sangat muda. Inti pesan gambar ini adalah urutan sepuluh langkah implementasi dari kiri ke kanan.

Table 6. Ringkasan hasil studi kasus.

Langkah	Operasi	Hasil
1	tentukan parameter	hash SHA-512, blok 128, output 64
2	implementasikan SHA-512	fungsi hash tersedia
3	prepare_key	array 128 byte ternormalisasi
4	inner state	hasher siap menerima pesan
5	update pesan	state berisi ipad + pesan
6	transformasi 0x6a	padding berubah ke opad
7	outer hash	tag 64 byte dihasilkan
8	verifikasi dengan RFC	tag cocok dengan test vector
9	verify constant-time	perbandingan tanpa kebocoran
10	zeroization pada drop	state terhapus saat tidak dipakai

11 Kesalahan Umum dan Cara Menghindarinya

Table 7. Kesalahan umum pada implementasi HMAC-SHA512.

Kesalahan	Akibat	Perbaikan
selalu hash kunci tanpa cek panjang	tag tidak cocok dengan standar	cek panjang, hash hanya jika > 128
tidak padding nol untuk kunci pendek	tag salah untuk kunci < 128	inisialisasi array dengan nol
ipad dan opad tertukar	tag valid untuk pesan lain	ipad = 0x36, opad = 0x5c
transformasi 0x6a diaplikasikan sebelum update	pesan rusak, tag salah	transformasi hanya di finalize
tidak zeroize block_key setelah new	kunci tertinggal di stack	panggil zeroize sebelum return
inner digest tidak di-zeroize	digest antara bisa bocor	bungkus dengan Zeroizing
perbandingan tag dengan == biasa	rentan timing attack	pakai XOR akumulasi constant-time
kunci panjang tanpa `Zeroizing` pada hash	digest antara tertinggal di stack	bungkus hash dalam Zeroizing
tidak ada `Drop` untuk HMAC	state tertinggal saat panic	implementasikan `Drop`
finalize memakai `&mut self`	state bisa dipakai ulang	finalize mengambil `self`
tidak menangani `None` dari `ih`	panic saat update ganda	pakai `unwrap_or_else`
tidak konsisten antara SHA-384 dan SHA-512	kode kompatibilitas rusak	pakai makro yang sama

Dua kesalahan yang paling sering muncul adalah selalu hash kunci tanpa cek panjang dan tidak zeroize `block_key` setelah konstruksi. Yang pertama membuat implementasi tidak kompatibel dengan standar. Yang kedua meninggalkan kunci di stack yang bisa bocor lewat memory disclosure.

12 Lembar Rujukan Cepat

Table 8. Lembar rujukan cepat HMAC-SHA512.

Kebutuhan	Formula atau detail
konstanta ipad	`0x36` diulang 128 kali
konstanta opad	`0x5c` diulang 128 kali
konstanta transformasi	`0x6a` = `0x36 XOR 0x5c`
ukuran blok SHA-512	128 byte
ukuran output SHA-512	64 byte
ukuran blok SHA-384	128 byte
ukuran output SHA-384	48 byte
ukuran blok SHA-256	64 byte
ukuran output SHA-256	32 byte
normalisasi kunci pendek	pad dengan `0x00` hingga 128 byte
normalisasi kunci tepat	pakai apa adanya
normalisasi kunci panjang	hash SHA-512 (64 byte), pad nol hingga 128 byte
inner hash	$H(\text{ipad} \oplus K' \parallel m)$
outer hash	$H(\text{opad} \oplus K' \parallel \text{inner})$
struktur HMAC	$(K' \oplus \text{opad}) \parallel H((K' \oplus \text{ipad}) \parallel m)$
verifikasi tag	XOR akumulasi seluruh byte
mask WASM	hash akumulasi sebagai lapisan tambahan
zeroization block_key	setelah inner hasher diinisialisasi
zeroization inner digest	setelah diserahkan ke outer hasher
zeroization padded	via `Drop` saat HMAC keluar scope
test vector kunci pendek	RFC 4231 Test Case 1
test vector kunci panjang	RFC 4231 Test Case 6

13 Kesimpulan

HMAC-SHA512 adalah konstruksi autentikasi pesan yang telah terbukti secara formal dan dipakai secara universal. Dua mekanisme kunci dalam implementasinya adalah normalisasi kunci dan transformasi `0x6a`. Normalisasi kunci memastikan bahwa kunci berapa pun panjangnya diproses menjadi array 128 byte yang konsisten. Transformasi `0x6a` memastikan bahwa padding inner bisa diubah menjadi padding outer tanpa menyimpan salinan kunci.

Tiga prinsip penutup:

1. Normalisasi kunci adalah persyaratan kompatibilitas. Implementasi yang tidak mengikuti tiga jalur `prepare_key` akan menghasilkan tag yang tidak cocok dengan implementasi lain. Ini adalah bug paling sulit dideteksi karena hanya muncul pada kunci dengan panjang tertentu.

2. Transformasi `0x6a` adalah konsekuensi aljabar. Konstanta `0x6a` bukan pilihan estetika, melainkan hasil XOR `0x36` dan `0x5c`. Memahaminya memungkinkan optimasi tanpa mengorbankan kompatibilitas.

3. Verifikasi tag wajib constant-time. Perbandingan naif dengan == membocorkan informasi prefix yang cukup untuk melakukan timing attack. Untuk sistem yang terekspos jaringan,

kebocoran ini bisa diselesaikan dalam hitungan detik.

Pengembang yang memahami ketiga prinsip ini dapat mengimplementasikan HMAC-SHA512 yang kompatibel dengan standar, atau mengaudit implementasi yang sudah ada dengan mata yang lebih kritis.

Referensi

References

[1] Krawczyk, H., Bellare, M., & Canetti, R. (1997). *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. IETF. https://www.rfc-editor.org/rfc/rfc2104

[2] Nystrom, M. (2005). *Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512*. RFC 4231. IETF. https://www.rfc-editor.org/rfc/rfc4231

[3] National Institute of Standards and Technology. (2015). *Secure Hash Standard (SHS)*. FIPS PUB 180-4. https://doi.org/10.6028/NIST.FIPS.180-4

[4] Bellare, M., Canetti, R., & Krawczyk, H. (1996). Keying Hash Functions for Message Authentication. In *Advances in Cryptology — CRYPTO '96* (pp. 1–15). Springer. https://doi.org/10.1007/3-540-68697-5_1

[5] Bellare, M. (2006). New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In *Advances in Cryptology — CRYPTO '06* (pp. 602–619). Springer. https://doi.org/10.1007/11818175_36

[6] The Rust Project Developers. (2026). *zeroize: Securely clear secrets from memory*. docs.rs. https://docs.rs/zeroize/

[7] The Rust Project Developers. (2026). *The Rustonomicon: The Dark Arts of Unsafe Rust*. https://doc.rust-lang.org/nomicon/

[8] The Rust Project Developers. (2026). *Rust API Guidelines*. https://rust-lang.github.io/api-guidelines/

[9] Khafidi, M. R. (2026). *libvctrl_sha512: SHA-512, HMAC-SHA512, HKDF-SHA512*. docs.rs. https://docs.rs/libvctrl_sha512

[10] Khafidi, M. R. (2026). *libvctrl: A Precision Toolkit for Building Custom Version Control Systems*. GitHub. https://github.com/mroczect/libvctrl

[11] Bernstein, D. J. (2005). *Understanding Brute Force*. Chapter 5 in *Introduction to Cryptography*. https://cr.yp.to/papers.html

[12] Kocher, P. C. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96* (pp. 104–113). Springer. https://doi.org/10.1007/3-540-68697-5_9

[13] Bernstein, D. J. (2005). Cache-Timing Attacks on AES. *Technical Report*. University of Illinois at Chicago. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[14] Percival, C. (2009). *Stronger Key Derivation via Sequential Memory-Hard Functions*. BSDCan 2009. https://www.tarsnap.com/scrypt/scrypt.pdf