Verifikasi Kriptografi Primitif: SHA-256

ANDREW W. APPEL, Universitas Princeton

Verifikasi formal lengkap yang diperiksa mesin dari program C: implementasi OpenSSL dari SHA-256. Ini adalah bukti interaktif kebenaran fungsional dalam asisten bukti Coq, menggunakan program C yang Dapat Diverifikasi logika. Verifiable C adalah logika pemisahan untuk bahasa C, terbukti benar sehubungan dengan semantik operasional untuk C, terhubung ke kompiler C pengoptimalan yang diverifikasi CompCert.

Kategori dan Deskripsi Subjek: D.2.4 **[Verifikasi Perangkat Lunak/Program]:** Bukti kebenaran; E.3 **[Enkripsi Data]:** Standar; F.3.1 **[Menentukan dan Memverifikasi serta Berpikir tentang Program]**

Ketentuan Umum: Verifikasi

1. PENDAHULUAN

> [K]riptografi sulit dilakukan dengan benar, dan satu-satunya cara untuk mengetahui apakah sesuatu dilakukan dengan benar adalah dengan memeriksanya. . . . Ini merupakan argumen yang sangat kuat untuk algoritma kriptografi sumber terbuka. . . . [Tapi] hanya dengan menerbitkan kode tidak secara otomatis berarti orang akan memeriksanya untuk keamanan kekurangan. Bruce Schneier [1999]
>
> Waspadalah terhadap perangkat lunak enkripsi komersial . . . [karena] kembali pintu... Cobalah untuk menggunakan enkripsi domain publik yang harus kompatibel dengan implementasi lainnya . . . Bruce Schneier [tahun 2013]

Yaitu, menggunakan implementasi sumber terbuka yang banyak digunakan dan diteliti dengan baik dari yang diterbitkan, algoritma standar yang tidak bersifat eksklusif, digunakan secara luas, dan telah diteliti dengan baik—karena “banyak “mata membuat semua serangga dangkal” hanya berhasil jika ada banyak mata yang memperhatikan.

Untuk ini saya tambahkan: gunakan implementasi yang diverifikasi secara formal dengan pemeriksaan mesin bukti kebenaran fungsional, resistensi saluran samping, sifat aliran informasi. “Banyak mata” adalah hal yang bagus, tetapi terkadang butuh waktu beberapa tahun untuk mengetahui bug [Bever 2014]. Verifikasi dapat menjamin properti program sebelum rilis secara luas.

Dalam makalah ini saya menyajikan langkah pertama: verifikasi formal atas kebenaran fungsional implementasi SHA-256 dari distribusi sumber terbuka OpenSSL.

Verifikasi formal tidak selalu dapat menggantikan jaminan banyak pihak. Misalnya, dalam kasus ini, saya hanya menyajikan jaminan kebenaran fungsional (dan akibat wajarnya, keamanan, termasuk tidak adanya buffer overruns). Mengenai properti lain seperti saluran sisi pengaturan waktu, saya tidak membuktikan apa pun; jadi sangat melegakan bahwa C yang sama ini Program ini telah digunakan dan diuji secara luas selama lebih dari satu dekade.

SHA-256, Secure Hash Algorithm dengan 256-bit digests, bukanlah algoritma enkripsi, tetapi digunakan dalam protokol enkripsi. Metode yang saya bahas dalam makalah ini dapat diaplikasikan pada masalah yang sama yang muncul dalam cipher seperti AES: interpretasi dokumen standar, protokol big-endian yang diimplementasikan pada mesin little-endian, sudut ganjil semantik C, penyimpanan byte dan pemuatan kata, ditandatangani dan aritmatika tak bertanda, aritmatika presisi yang diperluas, keandalan kompiler C, penggunaan instruksi khusus yang bergantung pada mesin untuk membuat segala sesuatunya lebih cepat, korespondensi model ke program, menilai basis terpercaya dari alat verifikasi.

Makalah ini menyajikan hasil berikut: Saya telah membuktikan kebenaran fungsional implementasi OpenSSL SHA-256, berkenaan dengan spesifikasi fungsional: formalisasi Standar Hash Aman FIPS 180-4 [FIPS 2012]. Pembuktian yang diperiksa mesin dilakukan menggunakan logika program C yang Dapat Diverifikasi, dalam asisten pembuktian Coq.
Bahasa C yang dapat diverifikasi terbukti benar berkenaan dengan semantik operasional C, dengan pembuktian yang diperiksa mesin dalam Coq. Program C dapat dikompilasi ke bahasa assembly x86 dengan kompiler C pengoptimalan yang diverifikasi CompCert; kompiler tersebut terbukti benar (dalam Coq) berkenaan dengan semantik operasional C yang sama dan semantik bahasa assembly x86. Jadi, dengan komposisi pembuktian yang diperiksa mesin tanpa celah, program bahasa assembly mengimplementasikan spesifikasi fungsional dengan benar.

Selain itu, saya menerapkan SHA-256 sebagai program fungsional di Coq dan membuktikannya setara dengan spesifikasi fungsional. Coq dapat menjalankan program fungsional pada string nyata (hanya sejuta kali lebih lambat daripada program C), dan mendapatkan jawaban yang sama seperti implementasi referensi standar.1 Hal ini memberikan keyakinan ekstra bahwa tidak ada hal konyol yang salah dengan spesifikasi fungsional.

Keterbatasan. Implementasinya berasal dari OpenSSL, dengan beberapa perluasan makro untuk membuatnya dari SHA-2 generik ke SHA-256. Saya memfaktorkan pernyataan penugasan sehingga paling banyak ada satu operan memori per perintah, misalnya, ctxÿh[0] += a; menjadi t=ctxÿh[0]; ctxÿh[0]=t+a; lihat §10.

CompCert menghasilkan bahasa assembly, bukan bahasa mesin; tidak ada bukti kebenaran assembler atau perangkat keras prosesor x86 yang dapat digunakan untuk menjalankan program yang dikompilasi. Asisten pembuktian Coq digunakan secara luas, dan kernelnya diyakini sebagai implementasi yang benar dari Kalkulus Predikatif Konstruksi Induktif (CiC), yang pada gilirannya diyakini konsisten.

Keterbatasan lain adalah pada waktu dan biaya untuk melakukan verifikasi. SHA-256 adalah "uji coba" untuk sistem Verifiable C. "Uji coba" ini mengungkap banyak ketidakefisienan sistem otomasi pembuktian Verifiable C: lambat, menghabiskan banyak memori dan sulit digunakan di beberapa tempat, dan tidak lengkap: beberapa sudut bahasa C tidak memiliki dukungan otomasi yang memadai. Namun ketidaklengkapan ini, atau ketidakstabilan otomasi pembuktian, tidak dapat membahayakan jaminan menyeluruh atas kebenaran logis yang diperiksa mesin: setiap langkah pembuktian diperiksa oleh kernel Coq.

Tidak terbatas. Cara lain yang membuat implementasi saya berbeda dari OpenSSL adalah saya menggunakan instruksi byte-swap x86 sehubungan dengan pemuatan/penyimpanan 4-byte big-endian (karena ini adalah mesin little-endian). Ini menggambarkan praktik umum saat mengimplementasikan primitif kriptografi pada mikroprosesor tujuan umum: gunakan instruksi khusus yang bergantung pada mesin untuk mendapatkan kinerja. Adalah baik bahwa logika program dapat bernalar tentang instruksi tersebut.

Bagaimana dengan menggunakan gcc atau LLVM untuk mengompilasi SHA-256? Untungnya, kompiler ini (gcc, LLVM, CompCert) cukup setuju dengan semantik C, jadi verifikasi SHA-256 masih dapat menambah kepastian bagi pengguna kompiler C lainnya. Di sebagian besar tempat yang jarang mereka tidak setujui, CompCert benar dan yang lainnya menunjukkan bug [Yang et al. 2012]; tidak ada bug yang pernah ditemukan dalam fase CompCert di balik Verifiable C.2

1Itu 0,25 detik per blok, dibandingkan 0,25 mikrodetik; cukup cepat untuk menguji spesifikasi. Program fungsional Coq sejuta kali lebih lambat karena mensimulasikan teori logika bilangan bulat biner yang digunakan dalam spesifikasi! Spesifikasi fungsional bahkan lebih lambat dari itu, karena fungsi W-nya mengambil faktor
4 waktu lagi.

2Artinya, CompCert memiliki fase front-end dari C ke C light; Verifiable C terpasang setelah fase ini, di C light. Yang et al. [2012] menemukan satu atau dua bug di fase front-end tersebut pada saat fase tersebut tidak diverifikasi secara formal, tetapi mereka tidak dapat menemukan bug apa pun di salah satu fase terverifikasi, yaitu antara C light dan bahasa assembly. Sejak saat itu, Leroy telah memverifikasi fase C-to-Clight secara formal, tetapi itu tidak menjadi masalah untuk Verifiable C, karena pada dasarnya kami memverifikasi kebenaran fungsional program C light. Selain itu, Yang

2. RANTAI ALAT PERANGKAT LUNAK YANG TERVERIFIKASI

Rantai Perkakas Perangkat Lunak Terverifikasi (VST) [Appel et al. 2014] memuat logika program C yang Dapat Diverifikasi untuk bahasa C, yang terbukti andal sehubungan dengan semantik operasional CompCert C. VST memiliki alat otomatisasi pembuktian untuk menerapkan logika program ini ke program C.

Salah satu gaya verifikasi formal perangkat lunak dilakukan dengan menerapkan logika program ke suatu program. Contoh logika program adalah logika Hoare, yang menghubungkan program c dengan spesifikasinya (prakondisi P, pascakondisi Q) melalui penilaian {P}c{Q}. Triple Hoare ini dapat dibuktikan dengan menggunakan aturan inferensi logika Hoare, seperti aturan komposisi sekuensial:

$$\frac{\{P\}\ c1\ \{T\}\ \{T\}\ c2\ \{R\}}{\{P\}\ c1;\ c2\ \{R\}}$$

Kami lebih memilih logika program atau algoritma analisis yang baik, yaitu, yang memiliki bukti bahwa apa pun yang diklaim logika program tentang program Anda benar-benar benar saat program dijalankan. VST terbukti baik dengan memberikan model semantik penilaian Hoare sehubungan dengan semantik operasional penuh CompCert C, sehingga kami benar-benar dapat mengatakan, apa yang Anda buktikan dalam Verifiable C adalah apa yang Anda dapatkan saat program sumber dijalankan. CompCert sendiri terbukti benar, jadi kami dapat mengatakan, apa yang Anda dapatkan saat program sumber dijalankan sama dengan saat program yang dikompilasi dijalankan. Dengan menggabungkan ketiga bukti ini: bukti program, kewajaran Verifiable C, dan kebenaran CompCert, kami mendapatkan: program yang dikompilasi memenuhi spesifikasinya.

Program C sulit diverifikasi karena seseorang perlu melacak banyak kondisi dan batasan sampingan: variabel ini diinisialisasi di sini, penambahan tersebut tidak meluap, p < q ini membandingkan pointer ke objek yang sama, pointer tersebut tidak menggantung. Logika C yang Dapat Diverifikasi melacak setiap hal ini; atau lebih tepatnya, membantu pengguna dalam melacak setiap hal. Kita tahu bahwa tidak ada asumsi yang terlewat, karena pembuktian kewajaran berkenaan dengan semantik C; dan kita tahu semantik C tidak melewatkan apa pun, karena pembuktian kebenaran CompCert berkenaan dengan eksekusi yang aman dalam bahasa assembly.

Tentu saja, ada cara yang lebih mudah untuk membuktikan bahwa program tersebut benar. Seseorang dapat menulis program fungsional dalam bahasa (seperti Gallina, ML, Haskell) dengan teori pembuktian yang jauh lebih bersih daripada C, dan kemudian upaya pembuktian menjadi lebih kecil menurut urutan besarnya. Setiap kali kinerja bahasa yang dikumpulkan dari sampah tingkat tinggi dapat ditoleransi, inilah cara yang tepat. Sejumlah besar perangkat lunak yang saat ini ditulis dalam Perl, Python, Javascript dapat ditulis ulang secara menguntungkan dalam bahasa fungsional dengan teori pembuktian yang bersih untuk verifikasi yang efektif. Namun, primitif kriptografi tidak ditulis dalam bahasa-bahasa ini; jika kita ingin memverifikasi implementasi kriptografi sumber terbuka yang mapan dan banyak digunakan, kita memerlukan perkakas untuk C.

Sintesis alih-alih verifikasi?. C tidak dirancang dengan mempertimbangkan teori pembuktian sederhana, jadi mungkin rute yang lebih sederhana untuk kripto yang terverifikasi adalah dengan menggunakan sintesis program dari bahasa spesifikasi khusus domain. Salah satu contohnya adalah Cryptol [Erkok et al. 2009] yang dapat menghasilkan C atau VHDL langsung dari spesifikasi fungsional. Pada prinsipnya, seseorang dapat berharap untuk membuktikan bahwa penyintesis Cryptol benar (meskipun ini belum dilakukan) atau memvalidasi output (yang mungkin lebih mudah daripada membuktikan program C tujuan umum).

---

dkk. [2012] menemukan bug spesifikasi di CompCert, terkait cara penanganan bit-field. Meskipun Leroy telah memperbaiki bug spesifikasi tersebut, hal ini juga tidak menjadi masalah: Verifiable C kebal terhadap bug spesifikasi di C, karena alasan yang dibahas di §8.

Sayangnya, bahasa sintesis terkadang memiliki ekspresi yang terbatas. Cryptol telah digunakan untuk mensintesis bagian block-shuffle dari SHA dari spesifikasi fungsional—tetapi hanya block-shuffle (fungsi yang disebut OpenSSL sha256 block data order).3 Menggunakan Verifiable C telah memverifikasi seluruh implementasi, termasuk padding, perhitungan panjang, penanganan multi-blok, pembaruan bertahap dari string yang tidak selaras, dan seterusnya. Synthesizer Cryptol (yang menerjemahkan Cryptol ke C atau VHDL) hanya dapat menangani blok berukuran tetap, sehingga tidak dapat menangani bagian-bagian ini (SHA256 Init, SHA256 Update, SHA256 Final, SHA256).

3. SPESIFIKASI SHA-256

Suatu program tanpa spesifikasi tidak mungkin salah, ia hanya bisa mengejutkan.4 Secara umum, seseorang dapat membuktikan suatu program C benar berkenaan dengan spesifikasi relasional. Misalnya, implementasi C yang menerapkan tabel pencarian harus memenuhi hubungan antara status program dan masukan/keluaran ini: jika pengikatan terkini untuk x adalah x ÿ y, maka pencarian x menghasilkan y.

Terkadang seseorang melakukan ini dalam dua tahap: membuktikan bahwa program C mengimplementasikan spesifikasi fungsional dengan benar (abstraksi implementasi), lalu membuktikan bahwa spesifikasi fungsional memenuhi spesifikasi relasional. Misalnya, implementasi C yang mengimplementasikan tabel pencarian dengan pohon pencarian biner seimbang mungkin terbukti benar sehubungan dengan spesifikasi fungsional pohon merah-hitam. Kemudian pohon merah-hitam fungsional dapat (lebih mudah) dibuktikan memiliki properti tabel pencarian.

Untuk hashing kriptografi, kami membuat spesifikasi fungsional dari standar FIPS 180-4 [FIPS 2012]. Spesifikasi relasionalnya adalah, "menerapkan fungsi acak." Sayangnya, tidak ada seorang pun di dunia yang tahu cara membuktikan bahwa SHA-256 menerapkan fungsi acak—bahkan di atas kertas—jadi saya tidak mencoba melakukan pembuktian yang diperiksa mesin (lihat §11).

FIPS 180-4 SHS (Secure Hash Standard) menyebutkan (dalam §3.2) aritmatika biner 32-bit tak bertanda dengan operator seperti penjumlahan-modulo-32, eksklusif-atau, dan pergeseran. Kita harus memberikan model aritmatika 32-bit dalam logika murni. Untungnya, Leroy telah mendefinisikan modul Integer tersebut dan membuktikan banyak propertinya sebagai bagian dari semantik CompCert C [Leroy 2009]; kita menggunakan ini secara langsung dalam spesifikasi fungsional, yang sepenuhnya independen dari bahasa C. Kita memiliki: tipe int; operasi seperti Int.add, Int.xor; injeksi (Int.repr : Z ÿ int) dari integer matematika ke integer 32-bit, dan proyeksi (Int.unsigned : int ÿ Z). Kita memiliki "aksioma" seperti,

$$\frac{0 \ddot{y} saya < 2^{32}}{\text{Int.unsigned (Int.repr i)} = i}$$

tetapi ini bukan aksioma dari logika yang mendasari (CiC), ini adalah teorema yang dibuktikan6 dari aksioma Coq menggunakan definisi konstruktif Int.unsigned dan Int.repr.

SHS mendefinisikan SHA-256 pada bitstring dengan panjang berapa pun, dan menjelaskan cara mengemasnya ke dalam bilangan bulat big-endian 32-bit. Implementasi OpenSSL mengizinkan urutan byte apa pun, yaitu kelipatan 8 bit. Kami merepresentasikan urutan nilai byte dengan urutan bilangan bulat matematika, dan kami dapat mendefinisikan fungsi pengemasan big-endian sebagai,

Definisi Z-ke-Int (abcd : Z) : Int.int :=
Int.atau (Int.atau (Int.atau (Int.shl (Int.repr a) (Int.repr 24)) (Int.shl (Int.repr b) (Int.repr 16)))

3Aaron Tomb, Galois.com, komunikasi pribadi, 13 Januari 2014.
4Parafrase dari JJ Horning, 1982.
5Stephen Yi-Hsien Lin menulis spesifikasi fungsional SHA-256 di Coq, yang kemudian saya adaptasi dan ditulis ulang.
6Selanjutnya, “terbukti” dapat dipahami berarti, “terbukti dengan bukti yang diperiksa secara mesin di Coq.”

(Int.shl (Int.repr c) (Int.repr 8))) (Int.repr d).

Diberikan daftar nl nilai byte (direpresentasikan sebagai bilangan bulat matematika, tipe Z), jika panjang nl merupakan kelipatan 4, mudah untuk mendefinisikan daftar bilangan bulat big-endian 32-bit yang sesuai:

Titik tetap Zlist-ke-intlist(nl: daftar Z): daftar int :=
cocokkan nl **dengan** h1::h2::h3::h4::t ÿ Z-ke-Int h1 h2 h3 h4 :: Zlist-ke-intlist t | -ÿ nihil

akhir.

Coq menggunakan Definisi untuk fungsi nonrekursif (atau nilai, atau tipe), dan Fixpoint untuk fungsi rekursif struktural. Operator :: adalah daftar cons. Operasi seperti Int.shl (geser ke kiri) dan Int.repr (representasi 32-bit dari integer matematika) diberi makna dasar oleh paket Int Leroy, seperti dijelaskan di atas. {256} ÿ 1

SHS mendefinisikan beberapa fungsi Ch, Maj, ROTR, SHR, ÿ contoh, {256} {256} {256} untuk dari 0 saya 1

Persamaan (x, y, z) = (x ÿ y) ÿ (¬x ÿ z)
Maj(x , y, z) = (x ÿ y) ÿ (x ÿ z) ÿ (y ÿ z)

Menerjemahkannya ke dalam Coq cukup mudah:

Definisi Ch (xyz : int) : int := Int.xor (Int.dan xy) (Int.dan (Int.bukan x) z).
Definisi Maj (xyz : int) : int := Int.xor (Int.xor(Int.dan xz)(Int.dan yz))(Int.dan xy).
Definisi Rotr bx : int := Int.ror x (Int.repr b).
Definisi Shr bx : int := Int.shru x (Int.repr b).
Definisi Sigma-0 (x : int) : int := Int.xor (Int.xor (Rotr 2 x) (Rotr 13 x)) (Rotr 22 x).
Definisi Sigma-1 (x : int) : int := ...
Definisi sigma-0 (x : int) : int := ...
Definisi sigma-1 (x : int) : int := ...

{256} {256}
Vektor K . . . K diberikan sebagai serangkaian konstanta heksadesimal 32-bit, karena 0 63 (0) (0) adalah vektor H . . . H 0 7 .

Di Coq kita menuliskannya dalam desimal, dan menyuntikkannya dengan Int.repr:

Definisi K := peta Int.repr [1116352408 , 1899447441, 3049323471, ..., 3329325298].

Definisi register awal := Map Int.repr [1779033703, 3144134277, ..., 1541459225].

Diberikan pesan M dengan panjang bit, SHS menjelaskan: tambahkan 1 bit, lalu cukup banyak bit nol sehingga pesan yang ditambahkan panjangnya akan menjadi kelipatan ukuran blok, lalu representasi 64-bit dari panjangnya. Karena kita memiliki pesan M dengan panjang n byte; kita tambahkan 128 byte (yang sudah memiliki 7 nol di belakangnya), lalu jumlah byte nol yang sesuai. Kita mengonversinya menjadi bilangan bulat 32-bit menggunakan big-endian, lalu menambahkan dua bilangan bulat 32-bit lagi yang mewakili bagian orde tinggi dan orde rendah dari panjang dalam bit.

Definisi generate-and-pad M := **let** n :=
Zlength M **dalam** Zlist-ke-
intlist (M ++ [128%Z] ++ list-repeat (Z.ke-nat (-(n + 9) mod 64)) 0)
++ [Int.repr (n ÿ 8 / Int.modulus), Int.repr (n ÿ 8)].

Perhatikan bahwa 0 ÿ a mod 64 < 64 meskipun a negatif. Angka ajaib 9 berasal dari 1+8: 1 byte terminator (nilai 128) ditambah 8 byte untuk bidang panjang 64-bit. Mengambil ÿ(n + 9) mod 64 memberikan jumlah byte padding yang diperlukan untuk membulatkan ke kelipatan 64 byte berikutnya, yang merupakan ukuran blok.

SHS mendefinisikan jadwal pesan Wt sebagai berikut:

Berat = (Saya) Gunung 0 ÿ t-15 = 0
{256} {256} (Wtÿ2) + Wtÿ7 + ÿ ... (Beratÿ15) + Beratÿ16 16 ÿ t ÿ 63 ÿ 1

di mana superskrip (i) menunjukkan nilai di blok pesan ke i. Kita menerjemahkannya ke dalam Coq sebagai,

```
Fungsi W (M: Z ÿint) (t: Z) {ukur Z.to-nat t} : int :=
  jika zlt t
  16 maka
  M t yang lain (Int.tambah (Int.tambah (sigma-1 (WM (t-2))) (WM (t-7)))
          (Int.tambahkan (sigma-0 (WM (t-15))) (WM (t-16)))).
```

Bukti.

```
intro; terapkan Z2Nat.inj-lt; omega. (ÿ t-2 < t ÿ) intro;
terapkan Z2Nat.inj-lt; omega. (ÿ t-7 < t ÿ) intro; terapkan
Z2Nat.inj-lt; omega. (ÿ t-15 < t ÿ) intro; terapkan Z2Nat.inj-
lt; omega. (ÿ t-16 < t ÿ)
```

Ditanya.

Coq adalah bahasa fungsi total. Pengukuran dan Proof/Qed menunjukkan bahwa fungsi W selalu berakhir. Ada satu baris pembuktian untuk masing-masing dari 4 pemanggilan rekursif; setiap pembuktian adalah, "memanggil nilai t yang lebih kecil."

Seseorang dapat menjalankan W ini sebagai program fungsional; tetapi memerlukan waktu eksponensial dalam t, karena ada 4 panggilan rekursif. Ia berfungsi dengan baik sebagai spesifikasi fungsional tetapi tidak dapat dieksekusi secara praktis.

Cipher blok menghitung hash 256-bit (8 kata) dari blok 512-bit (16 kata). Hash yang terakumulasi (i) (i) dari blok i adalah vektor H 7 . Untuk hash blok berikutnya, 0 . . . H delapan “variabel kerja” pertama hingga h diinisialisasi dari vektor H (i) .

64
iterasi fungsi Round ini dieksekusi:

```
Definisi register := daftar int.

Fungsi Putaran (regs: register) (M: Z ÿint) (t: Z)
      {ukuran (fun t ÿ Z.to-nat(t+1)) t} : register := jika zlt t 0 maka
 reg jika tidak cocok
 Bulatkan reg M (t-1) dengan | [a,b,c,d,e,f,g,h]
  ÿ
    biarkan T1 := Int.tambah(Int.tambah(Int.tambah(Int.tambah h (Sigma-1 e))(Ch efg))(nthi K t))(WM t) dalam
    biarkan T2 := Int.tambah (Sigma-0 a) (Maj abc)
     dalam [Int.tambah T1 T2, a, b, c, Int.tambah d T1, e,
  f, g] | -ÿ
  nihil akhir.
```

Bukti. intro; terapkan Z2Nat.inj-lt; omega. **Qed.**

Artinya, seseorang menyebut (Round r (blok nthi) 63). (Fungsi nthi bi mengembalikan elemen i dari daftar b, atau elemen sembarang Int.zero jika i negatif atau melebihi panjang daftar.) Jika panjang daftar regs bukan 8, hasil sembarang (daftar kosong) dikembalikan; tetapi hasilnya akan menjadi 8.

Saya merepresentasikan register sebagai daftar, bukan vektor yang diketik secara dependen (yaitu, daftar yang tipenya secara inheren memberlakukan pembatasan panjang) untuk menjaga spesifikasi sebagai urutan pertama sebisa mungkin. Ini menyederhanakan penalaran tentang spesifikasi, terutama portabilitasnya ke logika lain.

Fungsi putaran mengembalikan register a untuk B Bahasa Indonesia h yang kemudian ditambahkan ke H (i) menghasilkan H (i+1):

Definisi hash-block (r: register) (blok: daftar int) : register := map2 Int.tambahkan
r (Bulatkan r (blok ke-n) 63).

Diberikan pesan dengan panjang 16k, fungsi berikut menghitung H (k) dengan menerapkan blok hash ke setiap blok 16 kata berikutnya:

```
Fungsi hash-blocks (r: registers) (msg: daftar int) {ukur panjang msg} : registers :=
  cocokkan pesan
  dengan |
  nil ÿ r | -ÿ hash-blocks (hash-block r (16 pesan pertama)) (lewati 16 pesan)
  akhir.
```

Bukti. ... Qed.

Terakhir, SHA-256 menghasilkan intisari pesan sebagai string 32-byte oleh big-endian konversi dari H (k) .

Definisi SHA-256 (str : daftar Z) : daftar Z := intlist-
to-Zlist (hash-blok init-register (hasilkan-dan-pad str)).

4. PROGRAM FUNGSIONAL

Seseorang dapat membuktikan bahwa suatu program memenuhi spesifikasi, tetapi bagaimana seseorang mengetahui bahwa spesifikasi tersebut ditulis dengan benar? Salah satu cara untuk memperoleh keyakinan terhadap spesifikasi tersebut adalah dengan menghitung hasilnya pada serangkaian contoh, yaitu dengan menjalankannya. Fungsi SHA-256 yang diberikan di atas sebenarnya merupakan spesifikasi yang dapat dieksekusi. Coq mengizinkan spesifikasi relasional (nonkonstruktif, proposisional) yang tidak dijalankan, tetapi juga mengizinkan spesifikasi yang sepenuhnya konstruktif seperti yang satu ini.

Akan tetapi, karena fungsi W bersifat eksponensial, tidak praktis untuk menjalankan program ini. Oleh karena itu, saya menulis program fungsional alternatif yang disebut SHA-256. Program ini dapat dijalankan langsung di dalam asisten pembuktian Coq pada input aktual; dibutuhkan waktu sekitar 0,25 detik per blok.

Kunci dari “efisiensi” ini adalah fungsi W harus mengingat fungsi sebelumnya results.7 Di sini msg adalah daftar terbalik Wtÿ1, Wtÿ2, Wtÿ3, ...W0:

```
Definisi Wnext (msg : list int) : int := cocokkan
  msg dengan |
  x1::x2::x3::x4::x5::x6::x7::x8::x9::x10::x11::x12::x13::x14::x15::x16::- ÿ (Int.add (Int.add
        (sigma-1 x2) x7) (Int.add (sigma-0 x15) x16))
  | - ÿ Int.zero (ÿ tidak mungkin ÿ) akhir.
```

Haruskah kita khawatir tentang kasus yang "mustahil"? Coq adalah bahasa fungsi total, jadi kita harus mengembalikan sesuatu di sini. Salah satu alasan kita tidak perlu khawatir adalah karena saya membuktikan bahwa kasus ini tidak dapat menyebabkan program SHA-256 menjadi salah. Yaitu, saya membuktikan kesetaraannya (menggunakan aksioma ekstensionalitas):

Lemma SHA-256'-persamaan: SHA-256' = SHA-256.

Selain itu, fakta bahwa SHA-256 memberikan jawaban yang benar—pada semua masukan yang saya coba—memungkinkan kita mengetahui bahwa SHA-256 juga benar pada masukan tersebut.

7Juga dalam program yang efisien ini fungsi generate-and-pad dilakukan secara sangat berbeda.

Bukti kesetaraan ini memakan waktu sekitar satu hari untuk dibangun; ini jauh lebih cepat daripada membangun bukti bahwa implementasi C mengimplementasikan spesifikasi fungsional dengan benar. Namun terkadang kita harus memprogram dalam C—untuk mendapatkan SHA yang berjalan dalam mikrodetik, bukan detik.

Alih-alih menghitung hasil di dalam Coq, seseorang dapat mengekstrak program sebagai program ML, dan mengompilasinya dengan kompiler OCaml. Ini akan menghasilkan hasil yang lebih cepat daripada Coq, tetapi lebih lambat daripada C. Untuk tujuan pengujian spesifikasi SHA, hal ini tidak diperlukan.

5. PENGANTAR VERIFIKASI C

Bahasa C yang Dapat Diverifikasi dan logika program merupakan bagian dari bahasa C Light CompCert. Setiap program C yang Dapat Diverifikasi merupakan program C yang legal, dan setiap program C dapat diekspresikan dalam C yang Dapat Diverifikasi hanya dengan transformasi program lokal, seperti menghilangkan efek samping dari ekspresi: a=(b+=2)+3; menjadi b+=2; a=b+3; (terkadang variabel lokal tambahan diperlukan untuk menampung hasil antara). Kompiler CompCert menerima (pada dasarnya) bahasa C secara penuh; kami menggunakan bagian bukan karena kekurangan CompCert, tetapi untuk mengakomodasi penalaran tentang program. Lebih mudah untuk bernalar tentang satu tugas pada satu waktu.

Logika pemisahan. Kami menggunakan varian logika Hoare yang dikenal sebagai logika pemisahan, yang lebih ekspresif terkait anti-aliasing pointer dan pemisahan struktur data.
Kita menulis {P} c {Q} yang berarti (kurang lebih), jika P berlaku sebelum c dijalankan, maka Q akan berlaku setelahnya. Dalam logika pemisahan kita, pernyataan P memiliki bagian spasial yang berhubungan dengan isi jejak memori tertentu, dan bagian lokal yang berhubungan dengan variabel program lokal, dan bagian proposisional yang berhubungan dengan variabel matematika. Contoh pernyataan spasial adalah,

$$\underset{[0,n)}{\text{susunan f (p)}} \qquad (n: Z, f: Z \ddot{y} V, p: V)$$

yang merepresentasikan array n elemen yang dimulai pada alamat p, yang elemen ke i-nya adalah f (i). Di sini f adalah fungsi total dari integer (matematis) ke nilai; kita abaikan domain f di luar [0, n). Nilai V dapat berupa integer 32-bit (Vint i), representasi integer matematika 32-bit (Vint (Int.repr z)), floating point (Vfloat f), pointer (Vptr bi) dengan basis b dan offset dalam blok i, atau nilai yang tidak terdefinisi/tidak diinisialisasi (Vundef).

Konstruktor array C yang dapat diverifikasi sebenarnya membutuhkan dua argumen lagi: pembagian izin ÿ yang menunjukkan hanya-baca, baca-tulis, dll.; dan tipe elemen bahasa C, seperti tipe karakter yang tidak ditandatangani,

Definisi tuchar := Tint I8 Unsigned noattr.

Misalkan kita memiliki dua array berbeda p, q dan kita menjalankan penugasan p[i]=q[j]; satu spesifikasi yang mungkin adalah ini:

$$\{0 \ddot{y} i < j < n \ddot{y} (\text{array (p) } \ddot{y} \text{ array (q)})\}\ [0,n)\ [0,n)\ t = q[j];\ p[i] = t;\ \underset{[i:=g_G^F j]}{f}\ \{0 \ddot{y} i < j < n \ddot{y} (\text{array (p) } \ddot{y} \text{ array (q)})\}$$

$$[0,n)\ [0,n) \qquad G$$

Aturan inferensi logika pemisahan lebih menyukai penalaran tentang satu muatan atau penyimpanan pada satu waktu, jadi saya telah membuat transformasi program lokal. Di sini saya berasumsi ada dua larik terpisah p dan q yang isinya masing-masing adalah f dan g. Anda dapat mengetahui bahwa keduanya terpisah karena operator ÿ memberlakukan hal ini. Karena keduanya terpisah, kita tahu (dalam kondisi pasca) bahwa q tidak berubah, yaitu, isinya masih g. (Jika programmer bermaksud agar p dan q mungkin tumpang tindih, seseorang akan menulis spesifikasi yang berbeda.)

Variabel program, nilai simbolik. Ini sedikit penyederhanaan: i, j, p, q adalah variabel program, bukan variabel logika. C yang dapat diverifikasi membedakannya; seseorang dapat menulis prasyarat “untuk nyata” sebagai, PROP (0 ÿ i < j < n; bagian yang dapat ditulis

ÿ1)
LOKAL(`(eq i) (id-evaluasi - i); `(eq j) (id-evaluasi - j); `(eq p) (id-evaluasi -p); `(eq q) (id-evaluasi -q))
SEP (`(array-di tuchar ÿ1 f 0 np); `(array-di tuchar ÿ2 g 0 nq))

di mana bagian PROP mempunyai proposisi logika murni (yang tidak merujuk ke status program); LOCAL memberikan pernyataan tentang variabel lokal dari status program (tetapi bukan memori); dan SEP merupakan konjungsi pemisah dari pernyataan spasial, yaitu, tentang berbagai bagian memori yang terpisah.

Notasi `(eq i) (eval-id - i) berarti, “variabel program C - i berisi nilai simbolik i. Ini secara efektif merupakan pernyataan tentang lingkungan variabel lokal status program saat ini ÿ. Notasi `f mengangkat f ke lingkungan variabel lokal [Appel et al.
[2014, Bab 21], yaitu, `(eq i) (eval-

id - i) = (fun ÿ ÿ (eq i) (eval-id - i ÿ)) = (fun ÿ ÿ (fun x ÿ i = x) (eval-id - i
ÿ)) = (fun ÿ ÿ i = eval-id - i ÿ).

atau dengan kata lain, mencari - i dalam ÿ menghasilkan i.

Izin. Array p harus dapat ditulis, sedangkan array q harus setidaknya hanya dapat dibaca. Hal ini dinyatakan dengan pembagian izin: pembagian izin p ÿ1 harus memenuhi predikat pembagian yang dapat ditulis. Kita tidak perlu mengatakan pembagian yang dapat dibaca ÿ2 karena hal itu tersirat oleh predikat array-at.

Kami menyebut pembagian izin ini dan bukan hanya izin karena dalam pengaturan memori bersama bersamaan, pembuktian dapat membagi ÿ1 = ÿ1a + ÿ1b menjadi pembagian yang lebih kecil yang diberikan ke utas bersamaan. Pembagian ÿ1a dan ÿ1b ini tidak akan cukup kuat untuk izin menulis, tetapi keduanya dapat cukup kuat untuk izin membaca. Itu mengizinkan protokol penulisan-bersama-baca-eksklusif. Sekarang, anggaplah SHA-256 dipanggil dalam satu utas program bersamaan. Parameternya (string yang akan di-hash) dapat berupa larik bersama yang hanya-baca, tetapi hasilnya (larik untuk menampung intisari pesan) harus dapat ditulis. Semua ini dinyatakan secara ringkas dalam anotasi pembagian izin dari spesifikasi SHA-256 saya.

Alur kontrol. Bahasa C memiliki alur kontrol: perintah c mungkin gagal secara normal, mungkin melanjutkan loop, atau memutus loop, atau kembali dari suatu fungsi. Jadi, postkondisi Q harus memiliki hingga empat pernyataan berbeda untuk kasus-kasus ini. Untuk kasus di mana semua kecuali gagal dilarang—yaitu, tiga dari empat pernyataan postkondisi ini adalah **Salah—gunakan** konstruksi normal-ret-assert.

```
normal-ret-tegaskan (
  PROP ()
  LOKAL(`(eq i) (id-evaluasi - i); `(eq j) (id-evaluasi - j); `(eq p) (id-evaluasi -p); `(eq q) (id-evaluasi -q))
  SEP (`(array-di tuchar ÿ1 (pembaruan fi (qj)) 0 np); `(array-di tuchar ÿ2 g 0 nq)))
```

Kondisi pasca ini menunjukkan bahwa array p telah berubah di satu titik dan q tidak berubah. Kita dapat menghilangkan (0 ÿ i < j < n) dari kondisi pasca, karena ini adalah fakta logis yang tidak bergantung pada keadaan, dan (jika benar dalam kondisi pra) adalah benar selamanya.

Penalaran tingkat tinggi. Logika pemisahan biasa tidak ekspresif mengenai penunjuk fungsi, abstraksi data, dan konkurensi; jadi Verifiable C adalah logika pemisahan konkuren impredikatif tingkat tinggi. Tingkat tinggi berarti seseorang dapat mengukur lebih dari predikat. Ini berguna untuk menentukan tipe data abstrak. Ini juga berguna untuk penunjuk fungsi: jika fungsi f mengambil parameter p yang merupakan penunjuk fungsi, maka

prasyarat f akan mencirikan spesifikasi p, yaitu prasyarat dan pasca-syarat p. Bila spesifikasi penunjuk fungsi digunakan untuk menggambarkan program berorientasi objek, maka kuantifikasi impredikatif atas spesifikasi ini diperlukan.

Seseorang mungkin berpikir bahwa C bukanlah bahasa berorientasi objek, tetapi pada kenyataannya programmer C sering menggunakan pola desain yang mereka ekspresikan dengan void ÿ. Sistem tipe C terlalu lemah untuk "membuktikan" bahwa semua cast void ÿ ini benar, tetapi kita dapat menentukan dan membuktikannya dengan logika program.

Verifikasi SHA tidak menggunakan fitur-fitur tingkat tinggi ini, meskipun seseorang dapat menggunakan abstraksi data untuk struktur konteks, SHA256state-st. Namun, OpenSSL menggunakan konstruksi "mesin" berorientasi objek untuk menyusun HMAC dengan SHA.

Bacaan lebih lanjut. Appel et al. [2014] memberikan penjelasan lengkap tentang logika program.

6. PROGRAM C

Implementasi SHA-256 pada OpenSSL cerdas dalam beberapa hal (banyak di antaranya yang disengaja dalam desain SHA-256):

(1) Bekerja dalam satu lintasan, menunggu hingga akhir sebelum menambahkan bantalan dan panjang.
(2) Memungkinkan hashing inkremental. Misalkan pesan yang akan di-hash tersedia, secara berurutan, dalam segmen s1, s2, . . . , sj . Seseorang memanggil SHA256 Init untuk menginisialisasi konteks, SHA256 Update dengan setiap si secara bergantian, kemudian SHA256 Final untuk menambahkan padding dan panjang serta melakukan hash pada blok terakhir. Jika si tidak selaras dengan blok, maka SHA256 Update mengingat blok parsial dalam buffer. Namun, blok internal ke salah satu si tidak diputar melalui buffer; fungsi urutan data blok sha256 beroperasi padanya secara langsung dari memori tempat ia diteruskan ke SHA256 Update.
(3) Dalam perhitungan 64 putaran, hanya 16 elemen terbaru yang disimpan dalam buffer yang diakses modulo 16 menggunakan bitwise-dan dalam subskrip array.
(4) Dalam menambahkan panjang si ke hitungan 64 bit yang terakumulasi, terdapat uji luapan: apakah hasil dari (a + b) mod 232 < a? Jika demikian, tambahkan carry ke kata orde tinggi. Uji semacam itu mudah salah [Wang et al. 2013]; di sini berhasil karena a, b dideklarasikan tidak bertanda, tetapi pembuktian tetap berharga.
(5) Dalam fungsi SHA256 Final, ada satu blok terakhir yang berisi 1-bit, padding, dan panjang. Namun, bisa saja ada dua blok “terakhir”, jika isi pesan berakhir dalam jarak 8 byte dari akhir blok (jadi tidak ada ruang untuk 1-bit ditambah panjang 64-bit).
(6) Status yang terakumulasi antara panggilan ke Pembaruan SHA256 disimpan dalam rekaman yang “dimiliki” oleh pemanggil dan diinisialisasi oleh Inisiatif SHA256. Namun, vektor W murni lokal untuk fungsi “round” (urutan data blok sha256), jadi disimpan sebagai array variabel lokal (dialokasikan tumpukan). Meskipun itu tidak terlalu pintar, itu terlalu pintar untuk beberapa sistem verifikasi bahasa C, yang (tidak seperti Verifiable C) tidak dapat menangani variabel lokal yang dapat dialamatkan [Greenaway et al. 2012; Carbonneaux et al. 2014].

Klien SHA-256 memanggilnya sebagai berikut:

```
typedef struct SHA256state-st { unsigned
    int h[8];                    // Vektor H unsigned
    int Nl,Nh; // Panjang, angka 64-bit dalam dua bagian unsigned char
    data[64]; // Blok parsial belum di-hash unsigned int num; // Panjang
    fragmen pesan
} SHA256-CTX;

SHA256-CTX c;
karakter digest[32];
karakter ÿm1, ÿm2, ..., ÿmk ;
int tak bertanda n1, n2, ..., nk ;
```

```
//Bagaimana pemanggil membuat hash pada sebuah pesan:
SHA256- Inisiatif(&c);
SHA256-Perbarui(&c, m1, n1);
SHA256-Perbarui(&c, m2, n2);
...
SHA256-Pembaruan(&c, mk , nk );
SHA256-Final(intisari, &c);
```

Rangkaian mi dengan panjang ni masing-masing membentuk pesan. Idenya adalah bahwa Init menyiapkan konteks c dengan status register awal ch[ ], lalu setiap Update meng-hash beberapa blok lagi ke dalam status register tersebut. Jika mi bukan blok penuh, atau lebih tepatnya jika j=1 nj bukan kelipatan ukuran blok, maka blok parsial disimpan dalam (disalin ke) konteks c. Kemudian panggilan (i + 1) ke Update akan menggunakan fragmen itu sebagai awal blok penuh berikutnya. mi tidak perlu terpisah; pemanggil dapat membangun bagian-bagian pesan yang berurutan dalam buffer m yang sama.

Setelah panggilan ke i, register ch[ ] berisi hash dari semua blok penuh yang telah terlihat sejauh ini, dan panjang Nl,Nh berisi panjang (dalam bit) dari semua fragmen pesan, yaitu, 8 · Pada akhirnya, panggilan Final menambahkan padding dan

ni adalah bilangan bulat positif .

panjang, dan melakukan hash pada blok terakhir.
Nilai ch[ ] terakhir kemudian dikembalikan sebagai byte-string, intisari pesan.

Sebagian besar pilihan implementasi yang "cerdas" ini tidak terlihat secara langsung dalam spesifikasi fungsional, dan tidak dapat direpresentasikan dalam bahasa khusus domain seperti Cryptol. Pilihan tersebut hanyalah pemrograman C untuk keperluan umum, dan bahasa spesifikasi kita harus dapat menalarnya.

7. MENENTUKAN PROGRAM C

Spesifikasi Logika Pemisahan dari program C menghubungkan program (dan struktur data dalam memorinya) dengan properti kebenaran fungsional atau relasional. Lampiran B memberikan spesifikasi logika pemisahan lengkap dari program OpenSSL SHA-256; berikut ini saya sajikan sebagiannya.

Struktur data SHA256-CTX memiliki makna konkret dan makna abstrak.
Arti konkretnya diberikan oleh 6 tupel nilai ini, yang sesuai dengan 6 bidang struktur:

Definisi s256state := (daftar val ÿ (val ÿ (val ÿ (daftar val ÿ val)))).
(ÿkomentar: h Nl Nh data jumlah ÿ)

Ada alasan khusus untuk menggunakan tupel di sini, alih-alih rekaman Coq: tipe tupel ini dihitung secara otomatis dari definisi struct bahasa C, di dalam logika kalkulasi Coq.

Makna abstraknya adalah bahwa semua blok penuh m1 + m2 + . . . + mi telah diurai8 menjadi urutan kata 32-bit yang kita sebut hash; dan fragmen kurang-dari-satu-blok yang tersisa adalah urutan byte yang kita sebut data.

8SHS menggunakan kata “parsed” untuk menunjukkan: pengelompokan byte/bit ke dalam kata-kata big-endian 32-bit, dan pengelompokan kata-kata 32-bit ke dalam blok-blok 16-kata.

```
Induktif s256abs :=                (ÿ Keadaan abstrak SHA-256 ÿ)
  S256abs: ÿ (hash: daftar int) (ÿ kata-kata yang di-hash, sejauh ini ÿ)
                (data: daftar Z), (ÿ byte dalam blok parsial ÿ) s256abs.
```

Notasi mewah ini sebenarnya hanya 2-tuple (hash,data); Saya mendefinisikannya dengan cara ini untuk memengaruhi nama yang dipilih Coq untuk variabel yang diperkenalkan.

Keadaan abstrak adalah abstraksi dari keadaan konkret. Saya membuat relasi ini formal dalam Coq sebagai berikut. Pertama, kita hitung berapa vektor H pada akhir hash:

```
Definisi s256a-regs (a: s256abs) : daftar int := cocokkan
  a dengan data hash S256abs ÿ hash-blocks init-registers hash akhir.
```

Perhatikan bahwa ini memanggil blok hash dari spesifikasi fungsional yang dijelaskan di bagian 3.

Berikutnya, kita dapat menghitung panjang bit dari kata-kata hash ditambah byte data:

```
Definisi s256a-len (a: s256abs) : Z :=
   cocokkan a dengan data hash S256abs ÿ (Zlength hashed ÿ 4 + Zlength data) ÿ 8 akhir.
```

Kita dapat mendefinisikan penggabungan 64-bit dari dua angka 32-bit, dan apa artinya bagi sebuah integer (matematika) yang dapat direpresentasikan dalam sebuah unsigned char:

```
Definisi hilo (hi: int) (lo:int) : Z := (Int.unsigned hi ÿ Int.modulus + Int.unsigned lo).
Definisi isbyteZ (i: Z) := (0 ÿ i < 256).
```

Akhirnya, berikut adalah hubungan abstraksinya:

```
Definisi s256- berhubungan (a: s256abs) (r: s256state) : Prop :=
   cocokkan dengan data hash S256abs ÿ s256-h r
      = petakan Vint (hash-blok init-register hash )
   ÿ (ÿ hai, ÿ lo, s256-Nh r = Vint hai ÿ s256-Nl r = Vint lo ÿ
            (Panjang Z hash ÿ 4 + data panjang Z)ÿ8 = hilo hi lo) ÿ s256-
   data r = peta Vint (peta Int.repr data) ÿ (panjang data <
   64 ÿ Untuk semua data isbyteZ) ÿ (16 | Panjang Z
   hash) ÿ s256-num r = Vint
   (Int.repr (data panjang Z)) akhir.
```

Dengan kata lain, keadaan konkret (rh, rNh, rNl, rdata, rnum) merepresentasikan keadaan abstrak a = (hashed, data) bila: — rh

adalah hasil hashing dari semua hashed; — panjang bit dari (hashed, data) sama dengan $rNh \cdot 2^{32} + rNl$; char rdata berkorespondensi secara tepat dengan deret integer (matematis) data; — panjang data lebih kecil dari ukuran blok,
dan setiap elemen data adalah 0 ÿ d < 256; — panjang hashed adalah kelipatan 16 kata; — panjang data adalah rnum byte.

Logika C yang dapat diverifikasi memiliki operator (data-at ÿ ÿ rp) yang menyatakan bahwa alamat memori p, ditafsirkan menurut tipe bahasa C ÿ berisi nilai data (struct/array/integer) r dengan izin akses ÿ. Misalnya: ÿ =t-struct-SHA256state-st, p adalah penunjuk ke struct SHA256state-st, r adalah nilai status konkret (rh, rNh, rNl, rdata, rnum), dan ÿ adalah izin akses penuh Tsh.

Untuk menghubungkan SHA256-CTX dalam memori ke status abstrak, kita cukup menyusun relasi data-at dan s256- relate:

```
Definisi sha256state- (a: s256abs) (c: val) : mpred := EX r : s256state,
    PROP (s256- relate
      ar )
      LOKAL ()
      SEP (data-di Tsh t-struct-SHA256state-st rc).
```

Hal ini menghubungkan a ke c dengan mengatakan bahwa terdapat suatu keadaan konkret r yang mana abstrak-ke-konkret tersusun dengan konkret-dalam-memori.

Pembaruan bertahap. Fungsi SHA256-Update memperbarui konteks c dengan byte data- dengan panjang len:

void SHA256_Update (SHA256_CTX *c, const void *data_, size_t len); Misalkan data-berisi urutan

bilangan bulat msg. Menambahkan msg ke status abstrak a = (hashed, oldfrag) menghasilkan status abstrak yang diperbarui a = (hashed++blocks, newfrag) saat, **Induktif** update-abs: list Z ÿs256abs ÿs256abs ÿProp := Update-abs: (ÿ msg hashed blocks oldfrag

newfrag, Zlength oldfrag < 64 ÿ Zlength newfrag < 64 ÿ (16 | Zlength hashed)
ÿ (16 | Zlength
blocks) ÿ oldfrag++msg = intlist-to-Zlist blocks
++ newfrag ÿ update-abs msg
(S256abs hashed oldfrag)
(S256abs (hashed++blocks)
newfrag)). di mana intlist-to-
Zlist membongkar kata-kata big-endian 32-bit ke dalam urutan
nilai byte.

Dengan pendahuluan yang telah didefinisikan, saya sekarang dapat menyajikan spesifikasi logika pemisahan. fungsi Pembaruan.

```
Definisi SHA256-Update-spec :=
DEKLARASIKAN -SHA256-Update
  DENGAN a: s256abs, data: daftar Z, c: val, d: val, sh: bagikan, len: nat PRE [ -
  c DARI tptr t-struct-SHA256state-st, -data- DARI tptr tvoid, - len DARI tuint ]
     PROP (len <= panjang data;
              s256a-len a + Z.of-nat len ÿ 8 < dua-p 64)
     LOKAL (`(eq c) (id-eval -c); `(eq d) (id-eval -data- ); `(eq (Z.dari-
                nat len)) (`Int.tidak ditandatangani(`paksa- int(id-eval - len))))
     SEP(`K-vektor (eval-var -K256 (tarray tuint 64));
           `(sha256state- ac); `(blok data sh data d))
  POST [ tayang ]
    EX a :s256abs,
     PROP (perbarui-abs (data len pertama) aa ) LOKAL ()
     SEP(`K-vektor (eval-var -K256 (tarray tuint 64));
           `(sha256state- ac); `(data-blok sh data d)).
```

DECLARE memberikan nama (pengidentifikasi fungsi bahasa C) dari fungsi yang sedang ditentukan. WITH mengikat variabel (logika/matematika) yang dapat digunakan baik pada kondisi awal maupun kondisi akhir.

Prasyaratnya memiliki bentuk PRE [x] PROP P LOCAL Q SEP R di mana x adalah parameter fungsi, diberi anotasi dengan tipe bahasa C-nya (misalnya, data- memiliki tipe pointer-to-void); P adalah PROPositions logika murni (yang tidak merujuk ke status input); Q adalah fakta lokal (yang tidak merujuk ke memori, tetapi dapat merujuk ke variabel program), dan R adalah fakta spasial (yang merujuk ke memori melalui logika SEParation).

Di sini, data adalah urutan bilangan bulat dan d adalah alamat dalam memori; keduanya adalah variabel logis. Klausa LOCAL menyatakan bahwa parameter data- fungsi tersebut sebenarnya berisi nilai d, parameter c berisi nilai penunjuk c, dan seterusnya. Klausa SEP menyebutkan tiga wilayah memori terpisah yang menarik: array global 64 kata K256; SHA256-CTX c yang merepresentasikan status abstrak a; dan blok data pada alamat d yang berisi data segmen pesan berikutnya.

Kondisi pasca diparameterisasi oleh nilai pengembalian (fungsi khusus ini tidak memiliki nilai pengembalian). Bagian PROP-nya menghubungkan a dengan status abstrak baru a ; bagian LOCAL kosong (karena tidak ada nilai pengembalian untuk dikarakterisasi), dan bagian spasial (SEP) mengatakan: array global K256 masih ada di sana tanpa perubahan; di alamat c sekarang ada status yang diperbarui a ; dan blok data di d masih ada di sana tanpa perubahan.

Akibatnya. Berdasarkan sifat logika pemisahan, spesifikasi fungsional ini secara inheren memberikan jaminan khusus tentang kerahasiaan, integritas, dan tidak adanya buffer overrun:

—Satu-satunya variabel atau struktur data yang dibaca atau ditulis adalah yang disebutkan dalam spesifikasi fungsi.
—Satu-satunya variabel atau struktur data yang ditulisi adalah variabel atau struktur data yang disebutkan dengan izin menulis dalam spesifikasi fungsi.
—Nilai yang tertulis terbatas pada apa yang diklaim spesifikasi.

Misalnya, klausa SEP pada SHA256-Update-spec hanya menyebutkan blok memori pada alamat -K256, c, dan d; dan izin berbagi sh (yang mengendalikan d) tidak disebutkan sebagai dapat ditulis; hal ini sangat membatasi tempat SHA256-Update dapat membaca dan menulis.

Alat analisis statis seperti Coverity tidak dapat membuktikan kebenaran fungsional, tetapi setidaknya secara prinsip alat tersebut dapat menemukan masalah keamanan dasar. Namun, “Coverity tidak menemukan cacat heartbleed ... cacat tersebut tetap membandel bahkan ketika mereka mengubah berbagai pengaturan analisis. Pada dasarnya, alur kontrol dan alur data antara soket read() tempat data yang buruk berasal dan memcpy() yang akhirnya buruk terlalu rumit.” [Regehr 2014]

SHA-256, khususnya fungsi Update yang menyalin fragmen array, berisi aliran kontrol dan aliran data yang tidak sepele yang mengarah ke memcpy(). Namun, verifikasi penuh yang dilaporkan dalam makalah ini harus menganalisis kontrol dan data secara menyeluruh, tidak peduli seberapa rumitnya; dan logika tingkat tinggi (CiC) menyediakan alat yang cukup ekspresif untuk melakukannya. Kita tahu tidak ada heartbleed di SHA.

8. APAKAH SPESIFIKASINYA TEPAT?

Program SHA-256 terdiri dari sekitar 235 baris kode C (termasuk baris kosong dan komentar yang jarang).

Spesifikasi FIPS 180-4 SHA terdiri dari 35 halaman teks dan matematika; bagian khusus untuk SHA-256 mungkin terdiri dari 16 halaman. Spesifikasi fungsional saya—"terjemahan" ini ke logika—adalah 169 baris Coq, tetapi bergantung pada pustaka untuk teori matematika bilangan bulat tak terbatas, bilangan bulat 32-bit, dan daftar, yang jika digabungkan jumlahnya lebih banyak.

Spesifikasi saya tentang bagaimana kode C berkorespondensi fungsi demi fungsi dengan spesifikasi fungsional membutuhkan 247 baris Coq. Bukti dalam Coq jauh lebih besar—lihat §9—tetapi tidak perlu dipercaya karena sudah diperiksa oleh mesin.

Apakah kita memperoleh sesuatu ketika spesifikasi program 235 baris adalah 169+247
Apakah lebih mudah untuk memahami (dan mempercayai) programnya, atau spesifikasinya?
Ada beberapa alasan mengapa spesifikasi itu berharga:

(1) Dapat dimanipulasi secara logis. Misalnya, dua karakterisasi fungsi W yang berbeda dapat dibuktikan ekuivalennya.

(2) Spesifikasi fungsional dapat dijalankan pada data uji. Dalam kasus ini, kita melakukannya secara tidak langsung namun pasti dengan membuktikan kesetaraannya dengan program fungsional yang dijalankan pada data uji.

(3) Bukti kebenaran SHA-256 terhubung langsung dengan bukti koreksi kompiler C. ketepatan, di dalam pembuktian teorema tanpa spesifikasi “celah.”

Poin terakhir cukup penting. Program C untuk SHA-256 mungkin hanya terdiri dari 235 baris, tetapi maknanya bergantung pada pemahaman semantik bahasa C. Bahkan jika benar bahwa (saat ini) bahasa C memiliki spesifikasi yang jelas dan dipahami dengan baik,9 spesifikasi tersebut lebih besar dari 235 baris. Sebaliknya, dalam Verified Software Toolchain, spesifikasi bahasa C merupakan antarmuka internal antara logika program dan kompiler CompCert.
Artinya, anggaplah demi argumen bahwa spesifikasi C CompCert "salah;" atau logika program C yang Dapat Diverifikasi "salah." Itu tetap tidak akan menjadi masalah: dengan menyusun bukti kebenaran SHA-256 (dalam logika program C yang Dapat Diverifikasi), dengan bukti kemantapan logika program, dengan bukti kebenaran CompCert, kita memperoleh bukti menyeluruh tentang perilaku input/output yang dapat diamati dari program bahasa assembly, terlepas dari spesifikasi internalnya.

Namun, poin pertama juga penting: nilai dari spesifikasi adalah bahwa seseorang dapat berinteraksi dengannya secara logis. Anda tidak perlu berasumsi bahwa terjemahan saya atas dokumen SHS ke dalam Coq sudah benar; Anda memiliki kesempatan untuk menguji propertinya secara matematis (dengan membuktikan teorema tentangnya) di proof assistant.

Spesifikasi fungsi block-data-order, Init, Update, dan Final agak rumit, karena cara mereka mendukung hashing inkremental. Salah satu cara matematis untuk memperoleh keyakinan dalam spesifikasi ini adalah dengan menyusunnya. Yaitu, fungsi C ini

```
void SHA256(const unsigned char *d, ukuran_t n, unsigned char *md) {
        SHA256_CTX c;
        SHA256_Init(&c);
        SHA256_Perbarui(&c,d,n);
        SHA256_Final(md,&c);
}
```

harus setara dengan hashing SHA-256 nonincremental. Kita dapat melihat ini dalam spesifikasinya: **Definisi**

```
SHA256-spec := DECLARE
  -SHA256 WITH d: val,
  len: Z, dsh: share, msh: share, data: list Z, md: val PRE [ -d OF tptr tuchar, -n
  OF tuint, -md OF tptr tuchar ]
     PROP (berbagi-tulis msh; Z.of-nat (panjang data) ÿ 8 < dua-p 64)
     LOKAL (`(eq d) (eval-id -d); `(eq (Z.of-nat (panjang data))) (`Int.unsigned (`force-int (eval-id -n))); `(eq md) (eval-id -md))

     SEP(`K-vektor (eval-var -K256 (tarray tuint 64)); `(blok data
           dsh data d); `(blok memori msh (Int.repr 32) md))
    POST [ tayang ]
     SEP(`K-vektor (eval-var -K256 (tarray tuint 64)); `(blok data
            dsh data d); `(blok data msh (data SHA-256) md)).
```

Ini menyatakan bahwa pemanggilan SHA256(d,n,md) akan mengisi intisari pesan md dengan hash sebagaimana dihitung oleh spesifikasi fungsional (data SHA-256), selama memori pada d memang berupa data, dan datanya kurang dari dua miliar gigabyte. Selain itu, K256 (global) harus diinisialisasi dengan benar sebelumnya, dan dijamin terpelihara

9dan bahkan jika benar bahwa para ahli kompiler memahami spesifikasi tersebut dengan cara yang sama seperti yang dipahami oleh para programmer, hal ini diragukan [Wang et al. 2013]

tidak berubah; data masukan harus ada dan tidak akan diubah; dan keluaran
area harus dapat ditulis. Akhirnya, tidak ada memori lain (kecuali untuk catatan aktivasi
disebut fungsi) akan dibaca atau ditulis; ini adalah jaminan implisit (tapi sangat nyata)
logika pemisahan.

Fakta bahwa SHA256 memenuhi spesifikasi (relatif) sederhana ini adalah bukti
kasus non-inkremental, bahwa semua spesifikasi fungsi lainnya "benar"—yaitu,
mereka menyusun dengan benar. Ini bukan bukti bahwa kasus inkremental (lebih dari satu panggilan)
untuk Memperbarui) ditentukan dengan benar, tetapi hal itu membantu membangun jaminan. Jaminan tentang
kasus tambahan bergantung pada kebenaran definisi SHA256-Update-spec.

9. BUKTINYA

Bukti program fungsional berkenaan dengan spesifikasi fungsional cukup ringkas:

Komponen Detik Garis	
1022	29 Lemma tentang spesifikasi fungsional
1202	12 Bukti kebenaran program fungsional
tahun 2424	Jumlah 41

Saya menunjukkan di sini ukuran bukti Coq, dan waktu bagi Coq untuk memeriksa bukti secara batch
mode. Komponen pertama (lema) dibagikan dengan pembuktian program C.

Bukti kebenaran saya terhadap program C cukup besar—6539 baris bukti, ditulis oleh
tangan dengan beberapa potong-dan-tempel. Ini juga sangat lambat untuk memeriksa. Dengan demikian,
sistem C Verifi-able saat ini harus dianggap sebagai implementasi prototipe (meskipun tidak seperti kebanyakan
prototipe yang memiliki bukti kebenaran yang diperiksa mesin).

Komponen Detik Garis	
1022	29 Lemma tentang spesifikasi fungsional
229	83 Bukti fungsi penambahan panjang
1640	625 sha256 blok data pesanan()
43	256 SHA256 Inisialisasi()
1682	800 Pembaruan SHA256()
1484	687 SHA256 Akhir()
58	91 SHA256()
6539	Jumlah 2571

Menulis 6500 baris untuk memverifikasi program ini terlalu banyak pekerjaan. Saya yakin itu
bukti saya kikuk dan tidak elegan di banyak tempat dan mungkin ada bukti 2000 baris
berjuang untuk keluar. Namun mungkin solusi sebenarnya di sini adalah dengan meningkatkan secara drastis
otomatisasi pembuktian dengan menggunakan algoritma pencarian pembuktian modern seperti teori modul
kepuasan (SMT). Eksperimen terbaru telah menggabungkan kepercayaan Coq
(kernel kecil memeriksa bukti) dengan kekuatan SMT (program C++ besar yang mengklaim
ketidakpuasan) dengan mengekspor saksi bukti dari pemecah SMT ke Coq [Besson et al.
[Armand dkk., 2011].

Pemeriksaan bukti membutuhkan waktu 2571 detik (43 menit) pada satu prosesor Intel
core i7 (pada 3,4 Ghz) dengan cache yang banyak dan ram 2GB. Multicore, berjalan jauh lebih cepat
menggunakan parallel make. Tetap saja, 43 menit terlalu lama untuk program sebesar ini. Sebagai batch
perintah itu mungkin dapat ditoleransi; masalahnya ada pada bukti interaktif, di mana hal itu mungkin
butuh waktu 2 menit untuk melewati satu baris kode C—ini tidak terlalu interaktif.

Coq biasanya tidak terlalu lambat; masalahnya adalah eksekusi simbolik. Komponen yang tercantum
pada baris pertama tabel di atas (lema tentang spesifikasi fungsional dll.) tidak mengandung
eksekusi simbolik—tidak ada penerapan logika program bahasa C dengan semua sisinya
kondisi. Komponen tersebut hanya membutuhkan waktu 29 detik dalam Coq.

Mengapa eksekusi simbolik program C sangat lambat? Kami menulis prototipe pembuktian interaktif untuk Verifiable C sebagai eksekutor simbolik yang digerakkan pengguna yang diprogram dalam bahasa Ltac dari Coq. Saat eksekusi simbolik berlangsung, pengguna sering kali memberikan bukti untuk langkah-langkah yang tidak dapat ditemukan buktinya oleh otomatisasi. Sejauh ini, tidak ada masalah—meskipun akan lebih baik untuk memiliki lebih banyak otomatisasi, sehingga interaksi pengguna lebih sedikit; itu pekerjaan di masa mendatang. Masalahnya adalah Coq membangun jejak bukti dari eksekusi simbolik—yaitu, setiap langkah analisis sesuai dengan struktur data yang menggambarkan istilah bukti. Lebih buruk lagi, saat istilah tersebut dibangun, istilah tersebut sebenarnya adalah penutupan fungsi (catatan aktivasi) yang (ketika dipanggil) akan membangun istilah bukti konkret. Karena setiap langkah eksekusi simbolik memeriksa lusinan kondisi, penutupan fungsi konstruksi bukti dapat menempati ratusan megabita. Karena saya menjalankan Coq 32-bit yang dibatasi hingga 2 gigabita dan dengan pengumpul sampah penyalinan, ini menghabiskan sebagian besar memori.

Salah satu solusi yang tersedia adalah menjalankan Coq 64-bit pada komputer desktop 32-gigabyte saya. Namun, saya ingin berpikir bahwa memverifikasi program sekecil SHA-256 dapat dilakukan dengan mudah pada laptop biasa. Dua solusi lain untuk masalah ini mungkin tersedia:

(1) Programkan eksekusi simbolik menggunakan refleksi komputasional. Yaitu, tulis program fungsional dalam Coq untuk melakukan eksekusi simbolik dari logika program C yang Dapat Diverifikasi, buktikan kebenarannya dalam Coq, lalu terapkan pada program C seperti SHA-256.
Selama pelaksanaan program tersebut, Coq tidak membangun jejak pembuktian. Proyek VST sudah mengerjakan pendekatan ini [Appel et al. 2014, Bab 25,46,47].
(2) Coq tidak perlu membangun istilah pembuktian sebagai struktur data. Asisten pembuktian LCF di Edinburgh pada tahun 1979 mendemonstrasikan sebuah teknik untuk menggunakan kernel pemeriksa pembuktian kecil yang dapat dipercaya menggunakan antarmuka tipe data abstrak daripada struktur data istilah pembuktian [Gordon et al. 1979]. Beberapa sistem modern seperti HOL dan Isabelle/HOL juga menggunakan pendekatan ADT ini. Menggunakan ADT sebagai ganti istilah pembuktian akan menyelesaikan masalah memori, tetapi akan memerlukan perubahan substansial di dalam Coq.

10. APAKAH BENAR-BENAR OPENSSL?

Verifiable C merupakan bagian dari bahasa C; transformasi program tertentu diperlukan sebelum menerapkan logika program. Oleh karena itu, saya memodifikasi implementasi OpenSSL dari SHA-256 dengan cara berikut:

(1) OpenSSL sangat dimakrokan sehingga file sumber yang sama menghasilkan SHA-224, SHA-256, dan instansiasi lainnya. Saya memperluas makro dan menyertakan file header secukupnya untuk mengkhususkan diri pada SHA-256.
(2) Verifiable C melarang efek samping di dalam subekspresi; Saya membaginya menjadi beberapa subekspresi terpisah. pernyataan.
(3) Verifiable C melarang referensi memori di dalam subekspresi, dan mengharuskan setiap pernyataan penugasan memiliki paling banyak satu referensi memori di tingkat atas. Hal ini memerlukan penulisan ulang lokal, sering kali dengan pengenalan variabel sementara.
(4) Prototipe Verifiable C saat ini memerlukan pernyataan return dan bukan fall-through di akhir fungsi, jadi saya menambahkan beberapa pernyataan return.

Dua yang pertama ditangani secara otomatis oleh kompiler sebelum menerapkan logika program, sehingga tidak memerlukan perubahan manual apa pun pada program. Yang ketiga juga dapat ditangani oleh kompiler tetapi saat ini belum. Yang terakhir akan diperbaiki dalam waktu dekat.

Namun, dengan membuat beberapa makro, saya membuat tugas pembuktian saya lebih mudah, dengan mengorbankan batasan-batasan. menerapkan generalisasi hasil saya pada kasus 256-bit.

11. MENYUSUN BUKTI INI DENGAN ORANG LAIN

Verifikasi komponen individual dapat mengalami masalah karena ukuran komponen spesifikasi bisa lebih besar dari ukuran program. Bukti yang diperiksa mesin menghapus program dari basis tepercaya, tetapi jika spesifikasinya sebesar program, apa yang kita peroleh? Jawabannya bisa ditemukan dalam komposisi sistem. Ketika kami menyusun bukti SHA-256 dengan bukti C yang Dapat Diverifikasi dengan CompCert buktinya, seluruh spesifikasi C keluar dari basis terpercaya (seperti dijelaskan pada §8).

Di ujung lain, kita harus menghubungkan SHA-256 dengan aplikasinya, misalnya di Protokol HMAC untuk otentikasi kriptografi. HMAC menggunakan protokol kriptografi fungsi hash (seperti SHA-256).

Klaim yang diinginkan: Implementasi tertentu A dari HMAC dalam bahasa C adalah kunci-fungsi pseudorandom (PRF) yang dipilih pada pesan.

Struktur bukti:

(1) Program C A, yang memanggil SHA-256, mengimplementasikan fungsi tersebut dengan benar spesifikasi HMAC. (Pekerjaan selanjutnya: Akan diformalkan dan dibuktikan menggunakan teknik yang mirip dengan yang dijelaskan dalam makalah ini.)

(2) Spesifikasi fungsional HMAC (diindeks dengan kunci yang dipilih secara acak) memberikan PRF, pro-menunjukkan bahwa primitif hash yang mendasarinya adalah konstruksi hash Merkle-Damgard yang diterapkan pada fungsi kompresi PRF. Bukti: Pekerjaan masa depan berdasarkan (misalnya) Gazi et al. [2014] atau Bellare (3) et al. [1996; 2006] tetapi sepenuhnya diformalkan dalam Coq.10

SHA-256 OpenSSL menerapkan spesifikasi fungsional SHA-256 dengan benar. (Makalah ini.)

(4) Spesifikasi fungsional SHA-256 adalah konstruksi hash Merkle-Damgard. (Dapat dibuktikan dari spesifikasi fungsional yang dijelaskan dalam makalah ini; pekerjaan masa depan.)

(5) Fungsi kompresi yang mendasari SHA-256 adalah PRF.11 Ups! Tidak ada yang tahu cara membuktikan bahwa fungsi kompresi SHA-256 adalah PRF. Untuk saat ini, dunia bertahan pada fakta bahwa tidak seorang pun tahu bagaimana (tanpa mengetahui kuncinya) membedakannya dari fungsi acak.

Jadi ini adalah rangkaian bukti dengan lubang. Untungnya, lubang itu hanya ada di tempat kripto simetris selalu memiliki lubang: properti kripto dari primitif kunci simetris. Lubang ini dibatasi sangat erat oleh spesifikasi fungsional SHA-256 (169 baris Coq) dan fungsi satu arah. Sisanya—bagian yang berantakan dalam C—adalah tidak dalam basis tepercaya, koneksinya dibuktikan dengan bukti yang diperiksa mesin.

Itu hampir benar—tetapi sebenarnya ada satu hal lagi. Pada akhirnya pengguna akhir memiliki untuk memanggil fungsi HMAC, dari program C. Pengguna tersebut akan memerlukan spesifikasi terkait dengan konvensi pemanggilan bahasa C. Dalam makalah ini saya telah menunjukkan apa itu spesifikasi terlihat seperti untuk SHA-256: definisi SHA256-spec di §8. Ini tidak terlalu besar atau rumit; spesifikasi HMAC akan serupa.

10Ada rumor bahwa hal seperti ini telah dilakukan di CertiCrypt, tetapi tidak ada publikasi yang menjelaskan Bukti CertiCrypt HMAC atau NMAC. Ada deskripsi singkat tentang bukti EasyCrypt NMAC di [Barthe [et al. 2012], namun (tidak seperti CertiCrypt) EasyCrypt bukanlah hal mendasar: “EasyCrypt dirancang sebagai front-end ke kerangka kerja CertiCrypt. . . . Sertifikasi tetap menjadi tujuan penting, meskipun proses produksi bukti mekanisme ini mungkin akan keluar dari sinkronisasi sementara dengan pengembangan EasyCrypt.” [Barthe et al. 2012] Ini Tampaknya EasyCrypt/CertiCrypt terus-menerus tidak sinkron sejak 2012.

11Lebih tepatnya: pembuktian HMAC Bellare [2006] mengharuskan fungsi hash yang mendasari H adalah konstruksi Merkle-Damgard pada fungsi bulat R(x, m) sehingga: (A) “keluarga ganda” R aman terhadap serangan kunci terkait, dan (B) R adalah fungsi pseudorandom (PRF). Artinya, diberikan kunci acak yang tidak diketahui x, secara komputasi sulit untuk membedakan ÿm.R(x, m) dari fungsi yang dipilih secara acak di atas m.

12. BASIS YANG TERPERCAYA

Jaminan kebenaran SHA-256 bergantung pada basis tepercaya, serangkaian spesifikasi dan implementasi yang harus benar agar pembuktian menjadi bermakna. Artinya, kita harus memercayai:

Kalkulus Konstruksi Induktif. Logika yang mendasari Coq harus konsisten agar pembuktian di dalamnya dapat dipercaya. Beberapa makalah yang telah direviu telah diterbitkan yang memberikan argumen konsistensi untuk versi logika ini, tetapi makalah-makalah ini belum sepenuhnya melacak logika tertentu yang diterapkan dalam Coq. Secara umum, CiC adalah logika yang lebih kuat dan lebih kompleks daripada beberapa pesaingnya seperti HOL dan LF, jadi ada lebih banyak hal yang dapat dipercaya di sini.

Aksioma. Bukti yang masuk akal dari Verifiable C menggunakan aksioma ekstensionalitas fungsional (bergantung) dan ekstensionalitas proposisional. Kedua aksioma ini konsisten dengan teori inti Coq, yaitu, ketika keduanya ditambahkan ke CiC, masih mustahil untuk membuktikan bahwa keduanya salah. Namun pada tahun 2013 ditemukan bahwa ekstensionalitas proposisional tidak masuk akal dalam Coq 8.4.12 Ini bukan masalah yang melekat pada konsistensi aksioma, melainkan bug dalam pemeriksaan terminasi Coq.13 Diharapkan bahwa rilis Coq dalam waktu dekat akan konsisten dengan ekstensionalitas fungsional dan proposisional.

Kernel Coq. Coq memiliki kernel yang menerapkan pemeriksaan bukti (pemeriksaan tipe) untuk CiC. Kita harus percaya bahwa kernel ini bebas dari bug. Kernel ini memiliki sekitar 10.000 hingga 11.000 baris kode ML.

Kompiler OCaml. Kompiler ML kernel Coq dikompilasi oleh kompiler OCaml, yang terdiri dari puluhan ribu baris kode. Kompiler tersebut dikompilasi sendiri, yang menyebabkan regresi tak terbatas yang tidak dapat sepenuhnya dipercaya [Thompson 1984].

Runtime OCaml. Coq, berjalan sebagai biner yang dikompilasi OCaml, dilayani oleh Sistem runtime OCaml dan pengumpul sampah, ditulis dalam C.

Spesifikasi fungsional SHA-256. Kita harus percaya bahwa saya telah mentranskripsikan standar FIPS 180-4 ke dalam Coq dengan benar. Namun, jika (di masa mendatang) kita melengkapi bukti kripto tentang spesifikasi fungsional, maka elemen ini akan hilang dari basis tepercaya, sampai batas tertentu.

Spesifikasi API SHA-256. Hubungan yang dimaksudkan antara spesifikasi fungsional dengan struktur data pada panggilan fungsi API, yang saya sebut sebagai “spesifikasi API”, harus benar, jika tidak, saya telah membuktikan hal yang salah.

Spesifikasi CompCert. Saya telah menjelaskan sebelumnya bahwa kita tidak perlu mempercayai spesifikasi bahasa C CompCert, karena spesifikasi tersebut “keluar” dari basis tepercaya ketika disusun dengan bukti keandalan logika program C yang Dapat Diverifikasi.
Namun kita perlu percaya bahwa spesifikasi CompCert tentang bahasa assembly Intel x86 (IA-32) adalah benar.

Assembler. Saat ini, belum ada assembler yang terbukti benar untuk CompCert. Transisi dari bahasa assembly ke bahasa mesin dilakukan oleh assembler dan linker GNU. Membuktikan kebenaran assembler cukup dapat dicapai [Wu et al. 2003].

Intel Core i7. Biner OCaml dan pengumpul sampah berjalan dalam bahasa mesin. Komputer saya adalah Intel Core i7, dan kita harus percaya bahwa Intel telah menerapkan arsitektur set instruksi dengan benar, karena dua alasan: Pertama, kita menjalankan Coq di dalamnya, sehingga memengaruhi kepercayaan dalam pemeriksaan bukti. Kedua, kita menjalankan SHA-256 di dalamnya,

12Daniel Schepler, Maxime Den´ es, Arthur Chargu eraud, “Propositional extensionality is inconsistent in Coq,” milis coq-club, 12 Desember 2013.

13“Hanya untuk meyakinkan semua orang yang memiliki perkembangan yang mengandalkan aksioma ini, masalahnya tampaknya tidak terlalu dalam. Ada sedikit ketidakkonsistenan dalam cara pemeriksa penjaga menangani hipotesis ketidakterjangkauan, tetapi itu seharusnya dapat diperbaiki dengan mudah tanpa terlalu banyak berdampak pada kontribusi yang ada (semoga saja).” Maxime Den´ es, milis coq-club, 12 Desember 2013.

dan oleh karena itu spesifikasi bahasa assembly adalah bagian dari asumsi
bukti kebenaran CompCert.

Ini adalah rantai kepercayaan yang panjang. Kita bisa melakukan yang lebih baik. Proyek kode Foundational Proof-Carrying memiliki basis kepercayaan kurang dari 3000 baris kode, termasuk aksioma, kernel pemeriksaan bukti, kompiler, runtime, spesifikasi fungsional, spesifikasi API, dan spesifikasi ISA [Wu et al. 2003] (dan menghindari paradoks Thompson dengan tidak memerlukan kompiler dalam basis tepercaya). Namun itu untuk proyek yang jauh kurang ambisius (keamanan alih-alih kebenaran) dalam logika yang jauh lebih lemah (LF alih-alih Coq). Menskalakan teknik basis tepercaya kecil tersebut ke VST tidak akan mudah.

13. PEKERJAAN TERKAIT DI KRIPTO

Kita dapat membandingkannya dengan pekerjaan sebelumnya pada beberapa dimensi:

Spesifikasi. Apakah ada spesifikasi fungsi program? Apakah spesifikasinya
ditulis dalam bahasa fungsional (atau relasional) murni, yang dapat dianalisis dalam
asisten pembuktian? (Yaitu, selain verifikasi bahwa implementasi memenuhi
spesifikasi fungsional, dapatkah seseorang menalar tentang spesifikasi fungsional itu sendiri?)
Implementasi. Adalah bukti tentang implementasi yang efisien (misalnya, dalam bentuk yang dikompilasi)
C atau Java), atau hanya tentang spesifikasi fungsional?
Fondasi. Apakah ada bukti yang diperiksa mesin secara menyeluruh dari fondasinya?
logika, bahwa kode assembly yang dihasilkan mengimplementasikan spesifikasi dengan benar,
tanpa mempercayai kompiler (atau setara dengan, tanpa mempercayai spesifikasi bahasa pemrograman)?
(Jika tidak ada implementasi, pertanyaan ini tidak berlaku)
bahkan berlaku.)
Otomatis. Apakah verifikator memeriksa atau mensintesis algoritma kripto tanpa
banyak (atau ada) masukan atau anotasi manusia yang interaktif (atau tertulis)?
Umum. Dapatkah verifikator menangani semua bagian dari algoritma kripto (seperti kode manajemen
dalam Pembaruan SHA256), atau hanya bagian-bagian yang jumlah bit inputnya
sudah diperbaiki dan putarannya dapat dibuka sepenuhnya?

Spesifikasi, implementasi, dasar, tidak otomatis, umum:. Pekerja~~an yang dijela~~skan dijelaskan dalam makalah ini.

Spesifikasi, implementasi, tidak mendasar, ot~~omatis, tidak umu~~m:. Smith dan Dill [2008] memverifikasi beberapa implementasi cipher blok yang ditulis dalam Java, berkenaan dengan spesifikasi fungsional yang ditulis dalam Java atau ACL2. Mereka dikompilasi menjadi kode byte, kemudian digunakan model subset dari JVM untuk menghasilkan kode garis lurus. Hal ini membuat mereka kebal untuk bug di javac, tetapi kompiler JIT (dari kode byte ke kode asli) belum diverifikasi (atau, ekuivalennya, spesifikasi JVM mereka tidak terverifikasi). Mereka membuktikan kode garis lurus yang setara dengan kode garis lurus dari spesifikasi fungsional. Verifikasi mereka sepenuhnya otomatis, menggunakan aturan penulisan ulang untuk menyederhanakan dan menormalkan ekspresi aritmatika—dengan aturan untuk banyak pola khusus yang terjadi dalam kode kripto, seperti penggabungan bitfield oleh shift-and-or. Setelah menulis ulang, mereka menggunakan pemecah SAT untuk membandingkan ekspresi yang dinormalkan. Metode Smith dan Dill hanya berlaku jika jumlah bit input tetap dan loop dapat dibuka sepenuhnya. Verifikator mereka mungkin berlaku untuk fungsi pengocokan blok SHA-256 (sha256-block-data-order), tetapi tentu saja tidak untuk kode manajemen (SHA256-Pembaruan).

Spesifikasi terbatas~~, implemen~~tasi, tidak mendasar, otomat~~is, tidak umum:.~~ Cryptol [Erkok et al. 2009] menghasilkan C atau VHDL langsung dari spesifikasi fungsional, di mana jumlah bit input ditetapkan dan loop dapat dibuka sepenuhnya.
Faktanya, Cryptol memang melakukan unroll loop dalam sha256-block-data-order, yang mengarah ke sebuah program yang 1,5x lebih cepat dari implementasi standar OpenSSL (ketika keduanya dikompilasi

oleh gcc dan berjalan pada Intel Core i7)14. Spesifikasinya dalam dialek Haskell, tidak dapat langsung disematkan dalam asisten pembuktian yang ada; synthesizer tidak diverifikasi.

Spesifikasi, implementasi, tidak mendasar, tidak otomatis, tidak umum:. Toma dan Borrione [2005] menggunakan ACL2 untuk membuktikan kebenaran implementasi VHDL dari Algoritma pengacakan blok SHA-1.

Tidak ada spesifikasi, implementasi, tidak mendasar, otomatis, umum:. Seseorang dapat menerapkan algoritma analisis statis ke program C untuk mempelajari apakah mereka memiliki bug keamanan memori seperti buffer overruns. Banyak analisis semacam itu tidak masuk akal (akan hilang beberapa bug) dan tidak lengkap (akan melaporkan positif palsu tentang program bebas bug). Bahkan Jadi, mereka bisa sangat berguna; namun mereka tidak mencoba membuktikan kebenaran fungsional berkenaan dengan suatu spesifikasi.

Pekerjaan pelengkap. Dalam makalah ini saya berkonsentrasi pada verifikasi bahwa implementasi memenuhi spesifikasi fungsionalnya. Pekerjaan pelengkap menetapkan properti spesifikasi fungsional. Misalnya, Duan et al. [2005] membuktikan properti spesifikasi fungsional beberapa algoritma enkripsi: bahwa dekripsi adalah kebalikan dari enkripsi. Lebih relevan dengan SHA-256, Backes et al. [2012] memverifikasi secara mekanis (dalam EasyCrypt) bahwa konstruksi Merkle-Damgard memiliki sifat keamanan tertentu. Bellare [1996; 2006] memberikan bukti pertama keamanan NMAC/HMAC (tanpa bukti yang diperiksa mesin); Gazi et al. [2014] membuktikan keamanan PRF dari NMAC/HMAC (tanpa bukti yang diperiksa mesin), berdasarkan asumsi yang lebih sedikit.

EasyCrypt. Almeida et al. [2013] menjelaskan penggunaan alat EasyCrypt mereka untuk memverifikasi keamanan implementasi skema enkripsi RSA-OAEP. Spesifikasi fungsional RSA-OAEP ditulis dalam EasyCrypt, yang kemudian memverifikasi properti keamanannya. Skrip Python yang belum diverifikasi menerjemahkan spesifikasi EasyCrypt ke (ekstensi dari) C; kemudian ekstensi CompCert mengkompilasinya ke bahasa assembly. Akhirnya, alat kebocoran memverifikasi bahwa program bahasa assembly tidak memiliki lebih dari kebocoran penghitung program daripada kode sumber, yaitu jejak program yang dikompilasi cabang bersyarat tidak lebih informatif bagi musuh daripada kode sumbernya.

Verifikator EasyCrypt tidak sepenuhnya mendasar; ini adalah program OCaml yang kebenarannya tidak terbukti. Penerjemahan dari EasyCrypt ke C tidak mendasar. penerjemahan dari bahasa C ke bahasa assembly adalah dasar, menggunakan CompCert. Program harus beroperasi pada blok data berukuran tetap.

Model kebocoran adalah Model Penghitung Program (jejak cabang bersyarat), dan ada pemeriksa dasar (yaitu, terbukti benar dalam Coq) bahwa program yang dikompilasi tidak membocorkan informasi jejak PC lebih banyak daripada program sumber. Namun bentuk lain kebocoran tidak dimodelkan. Secara khusus, SHA-256 memiliki baris kode (ditandai /* biarkan tetap nol */) yang seluruh tujuannya adalah untuk mengurangi kebocoran fragmen pesan melalui memori yang tidak dialokasikan; tetapi saluran kebocoran semacam itu tidak dimodelkan di EasyCrypt.

Kode C EasyCrypt bergantung pada fungsi pustaka bignum yang dipanggil melalui bahasa nonstandar spesifikasi antarmuka—nonstandar, karena CompCert standar (melalui versi saat ini, 2.3) tidak memiliki cara untuk menangani panggilan fungsi eksternal yang menerima atau mengembalikan menghasilkan memori. Namun EasyCrypt tidak menyediakan mekanisme yang memungkinkan fungsi-fungsi ini dapat dibuktikan kebenarannya, juga tidak memberikan teori pembuktian mekanis untuk antarmuka spesifikasi khusus ini.

Singkatnya, Almeida et al. menyerang dua masalah yang saling melengkapi untuk pekerjaan yang telah saya lakukan. EasyCrypt memungkinkan penalaran tentang properti kripto

14Aaron Tomb, Galois.com, komunikasi pribadi, 13 Januari 2014.

spesifikasi fungsional; dan mereka memperluas jaminan kebenaran kompilasi CompCert dengan jaminan tentang saluran samping tertentu. Mereka tidak bernalar tentang hubungan semantik antara spesifikasi fungsional dan program C (baik algoritma utama maupun pustaka bignum).

14. PEKERJAAN TERKAIT DALAM VERIFIKASI C

Ada banyak alat analisis program untuk C. Kebanyakan dari alat-alat tersebut tidak membahas spesifikasi fungsional atau kebenaran fungsional, dan kebanyakan tidak tepat dan tidak lengkap.
Meskipun demikian, alat-alat tersebut sangat berguna dalam praktik: alat analisis statik C merupakan industri bernilai miliaran dolar
per tahun.15 Verifikasi formal mendasar dari program C baru-baru ini dimungkinkan. Karya-karya yang paling signifikan adalah kernel sistem operasi: seL4 [Klein et al. 2009] dan CertiKOS [Gu et al. 2011]. Kedua pembuktian tersebut merupakan pembuktian penyempurnaan antara spesifikasi fungsional dan semantik operasional. Kedua pembuktian tersebut dilakukan dalam logika tingkat tinggi: seL4 dalam Isabelle/HOL dan CertiKOS dalam Coq. Masing-masing proyek ini memverifikasi program C yang jauh lebih besar daripada program SHA-256 yang saya jelaskan di sini.

Tak satu pun dari kerangka kerja pembuktian mereka menggunakan logika pemisahan, tak satu pun dapat mengakomodasi penggunaan variabel lokal yang dapat dialamatkan dalam C, dan tak satu pun dapat menangani penunjuk fungsi atau spesifikasi tingkat tinggi. Ini berarti bahwa program OpenSSL SHA-256 tidak dapat dibuktikan dalam kerangka kerja ini, karena menggunakan variabel lokal yang dapat dialamatkan. Namun, penyesuaian kecil pada program C—memindahkan larik X ke dalam struktur SHA256state-st—akan menghilangkan penggunaan variabel lokal yang dapat dialamatkan.

SHA-256 tidak menggunakan penunjuk fungsi. Namun, OpenSSL menggunakan penunjuk fungsi dalam mekanisme "mesinnya", gaya pemrograman berorientasi objek yang menghubungkan komponen secara dinamis—misalnya, HMAC dan SHA. Logika program C yang dapat diverifikasi, dengan logika pemisahan tingkat tinggi, mampu menalar pola berorientasi objek tersebut dalam C [Appel et al. 2014, Bab 29].

Semantik C yang digunakan dalam pembuktian seL4 asli tidak terhubung ke kompiler C (misalnya, bukan semantik C CompCert), jadi seluruh semantik C merupakan bagian dari basis tepercaya pembuktian seL4. Pekerjaan yang lebih baru menghapus C dari basis tepercaya: Sewell dkk. [2013] melakukan validasi translasi untuk gcc 4.5.1 dengan mendekompilasi kode ARM menjadi grafik logis, kemudian menggunakan kombinasi pencarian pembuktian heuristik yang disetel dengan cermat dan penyelesaian SMT untuk menemukan pembuktian kesetaraan antara program sumber dan bahasa mesin.

CertiKOS terbukti benar sehubungan dengan semantik CompCert C, jadi (seperti dalam saya Bukti SHA-256) semantik C ini keluar dari basis tepercaya.

Baik seL4 maupun CertiKOS adalah program C yang baru ditulis dan dirancang untuk verifikasi. Pada prinsipnya, ini adalah cara yang tepat untuk melakukan berbagai hal: mungkin sulit untuk memverifikasi program yang sudah ada sebelumnya. Namun, ada kalanya penting untuk dapat melakukannya. Produsen pesawat, yang memiliki basis kode yang telah disertifikasi (dan dipercaya) untuk digunakan dalam jet penumpang, tidak boleh diminta untuk menulis ulang perangkat lunak fly-by-wire mereka hanya agar mereka dapat menerapkan teknik verifikasi yang baru dan lebih baik. Dan, sejauh komunitas keamanan telah mempercayai properti OpenSSL yang tidak berfungsi—kurangnya saluran pengaturan waktu, ketahanan terhadap penyuntikan kesalahan, kompatibilitas dengan banyak kompiler C—kepercayaan ini tidak serta merta dapat ditransfer ke implementasi baru.

15. KESIMPULAN

Verifikasi kebenaran fungsional program C memiliki aplikasi penting dalam keamanan komputer. Kebenaran memiliki konsekuensi keamanan memori, yang sangat berharga dalam

15Andy Chou, “From the Trenches: Static Analysis in Industry”, diundang berbicara di POPL 2014, 24 Januari 2014.

Namun dalam penerapan mekanisme perlindungan (seperti sistem operasi, enkripsi, otentikasi), keamanan saja tidak cukup: kebenaran adalah yang menjamin
bahwa mekanisme ini benar-benar mengamankan sistem yang seharusnya dilindungi.

Bahasa C tidak ramah terhadap verifikasi program: bahasa ini memiliki sudut-sudut yang rumit, seseorang perlu melacaknya dari banyak kondisi sampingan. Meskipun demikian, verifikasi formal penuh C dapat dilakukan program. Hasil sebelumnya telah menunjukkan hal ini untuk mikrokernel sistem operasi [Klein et al. 2009] (meskipun tidak dengan koneksi terverifikasi ke kompiler terverifikasi). hasil seperti itu, program C biasanya dibangun kembali dengan desain khusus cocok untuk tugas verifikasi.

Dalam proyek ini saya menunjukkan bahwa seseorang dapat memverifikasi sebuah program sebagaimana adanya. berharga karena primitif kriptografi sumber terbuka yang banyak digunakan memiliki banyak properti yang relevan selain kebenaran fungsional. Misalnya, komentar /* biarkan tetap nol */ dalam fungsi Pembaruan dilampirkan ke baris yang tidak memiliki dampak fungsional, tetapi mungkin mengurangi aliran informasi melalui variabel yang tidak dialokasikan. logika program tidak dapat membuktikan bahwa garis ini mengurangi saluran samping, tetapi setidaknya saya dapat membuktikan bahwa hal itu tidak mengganggu kebenaran fungsional. Apapun jaminan dan keyakinan yang diberikan yang diperoleh masyarakat dalam program ini hanya akan bertambah berkat verifikasi ini.

Pada saat yang sama, primitif kriptografi ini memiliki banyak varian implementasi (menggunakan instruksi yang bergantung pada mesin). Variasi kecil dari bukti ini dapat berfungsi untuk membuktikan semuanya setara dengan spesifikasi fungsional yang sama, bahkan jika ada tidak banyak mata yang mengawasi setiap bug untuk menjaga bug tetap dangkal.

REFERENSI

Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, dan Franc¸ois Dupressoir. 2013. Kriptografi berbantuan komputer bersertifikat: kode mesin yang efisien dan aman dari implementasi tingkat tinggi. Dalam Prosiding Konferensi ACM SIGSAC 2013 tentang Keamanan Komputer dan Komunikasi. ACM, 1217–
Tahun 1230.

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, dan Xavier Leroy. 2014. Logika Program untuk Kompiler Bersertifikat. Cambridge.

Michael Armand, Germain Faure, Benjamin Gregoire, Chantal Keller, Laurent Th ery, dan Benjamin Werner. 2011. Integrasi Modular Pemecah SAT/SMT ke Coq melalui Saksi Pembuktian. Dalam Bagian Pertama Konferensi Internasional tentang Program dan Bukti Bersertifikat.

Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Gregoire, Cesar Kunz, Malte Skoruppa, dan Santiago Zanella Beguelin. 2012. Keamanan terverifikasi Merkle-Damgarth. Dalam Dasar-dasar Keamanan Komputer Simposium (CSF), IEEE 25 tahun 2012. IEEE, 354–368.

Gilles Barthe, Juan Manuel Crespo, Benjamin Gregoire, Cesar Kunz, dan Santiago Zanella B eguelin. 2012. Bukti kriptografi berbantuan komputer. Dalam Pembuktian Teorema Interaktif. Springer, 11–27.

Mihir Bellare. 2006. Bukti baru untuk NMAC dan HMAC: Keamanan tanpa ketahanan terhadap benturan. Dalam Advances dalam Kriptologi-KRIPTO 2006. Springer, 602–619.

Mihir Bellare, Ran Canetti, dan Hugo Krawczyk. 1996. Mengunci fungsi hash untuk otentikasi pesan. Dalam Kemajuan dalam KriptologiCRYPTO96. Springer, 1–15.

Fred´eric Besson, Pierre-Emmanuel Cornilleau, dan David Pichardie. 2011. Bukti SMT Modular untuk Pemeriksaan Refleksif Cepat di dalam Coq. Dalam Konferensi Internasional Pertama tentang Program dan Bukti Bersertifikat.

Lindsey Bever. 2014. Bug besar yang disebut 'Heartbleed' mengungkap data Internet. Washington Post (9 April 2014).

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, dan Zhong Shao. 2014. Verifikasi Ujung-ke-Ujung Batas Ruang Tumpukan untuk Program C. Dalam Prosiding Konferensi ACM SIGPLAN 2014 tentang Desain dan Implementasi Bahasa Pemrograman (PLDI'14).

Jianjun Duan, Joe Hurd, Guodong Li, Scott Owens, Konrad Slind, dan Junxing Zhang. 2005. Fungsional pembuktian kebenaran algoritma enkripsi. Dalam Logika untuk Pemrograman, Kecerdasan Buatan, dan Penalaran. Springer, 519–533.

L. Erkok, Magnus Carlsson, dan Adam Wick. 2009. Ko-verifikasi perangkat keras/perangkat lunak algoritma kriptografi menggunakan Cryptol. Dalam Metode Formal dalam Desain Berbantuan Komputer, 2009 (FMCAD'09). IEEE, 188–191.

FIPS 2012. Secure Hash Standard (SHS). Laporan Teknis FIPS PUB 180-4. Laboratorium Teknologi Informasi, Institut Nasional Standar dan Teknologi, Gaithersburg, MD.

Peter Gazi, Krzysztof Pietrzak, dan Michal Rybar. 2014. Keamanan PRF yang Tepat untuk NMAC dan HMAC. Dalam Kemajuan dalam Kriptologi–CRYPTO 2014. Springer, 113–130.

Michael J. Gordon, Robin Milner, dan Christopher P. Wadsworth. 1979. Edinburgh LCF: Logika Komputasi yang Dimekanisasi. Catatan Kuliah dalam Ilmu Komputer, Vol. 78. Springer-Verlag, New York.

David Greenaway, June Andronick, dan Gerwin Klein. 2012. Menjembatani kesenjangan: Abstraksi terverifikasi otomatis Pembuktian Teorema Interaktif. Springer, 99–115.

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, dan David Costanzo. 2011. CertiKOS: Kernel Bersertifikat untuk Komputasi Awan yang Aman. Dalam Prosiding Lokakarya Sistem Asia-Pasifik Kedua (APSys'11). ACM, Artikel 3, 5 halaman. DOI: http://dx.doi.org/10.1145/2103799.2103803

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, dan lain-lain. 2009. seL4: Verifikasi formal kernel OS. Dalam Prosiding simposium ACM SIGOPS ke-22 tentang Prinsip-prinsip sistem operasi. ACM, 207–220.

Xavier Leroy. 2009. Kompiler Back-end yang Terverifikasi Secara Formal. Jurnal Penalaran Otomatis 43, 4 (2009), 363–446.

John Regehr. 2014. Perkembangan Baru untuk Coverity dan Heartbleed. Embedded in Academia (12 April 2014). blog.regehr.org/archives/1128 Bruce

Schneier. 1999. Open Source dan Keamanan. Buletin Crypto-Gram (15 September 1999).

Bruce Schneier. 2013. Cara Tetap Aman terhadap NSA. Buletin Crypto-Gram (15 September 2013).

Thomas AL Sewell, Magnus O. Myreen, dan Gerwin Klein. 2013. Validasi terjemahan untuk OS yang terverifikasi kernel. Pemberitahuan ACM SIGPLAN 48, 6 (2013), 471–482.

Eric W. Smith dan David L. Dill. 2008. Verifikasi formal otomatis dari implementasi cipher blok. Metode Formal dalam Desain Berbantuan Komputer (FMCAD'08). IEEE, 1–7.

Ken Thompson. 1984. Refleksi tentang Kepercayaan. 27, 8 (1984), 761–763.

Diana Toma dan Dominique Borrione. 2005. Verifikasi formal inti sirkuit SHA-1 menggunakan ACL2. Pembuktian Teorema dalam Logika Tingkat Tinggi. Springer, 326–341.

Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, dan Armando Solar-Lezama. 2013. Menuju sistem yang aman untuk pengoptimalan: menganalisis dampak perilaku yang tidak terdefinisi. Dalam Prosiding Simposium ACM ke-24 tentang Prinsip Sistem Operasi. ACM, 260–275.

Dinghao Wu, Andrew W. Appel, dan Aaron Stump. 2003. Pemeriksa Bukti Fondasi dengan Saksi Kecil. Dalam PPDP. 264–274.

Xuejun Yang, Yang Chen, Eric Eide, dan John Regehr. 2012. Menemukan dan memahami bug dalam kompiler C. Pemberitahuan ACM SIGPLAN 47, 6 (2012), 283–294.

LAMPIRAN

A. SHA-256, PROGRAM C YANG DISESUAIKAN DARI OPENSSL /*


```
eksternal unsigned int __builtin_read32_reversed (const unsigned int * ptr); eksternal void
__builtin_write32_reversed (unsigned int * ptr, unsigned int x);

#include <stddef.h> #include
<string.h> /* untuk memcpy, memset */

#definisikan HOST_c2l(c,l) \
      (l=(panjang tak bertanda)(__builtin_read32_terbalik (((int tak bertanda *)c))),c+=4,l)

#tentukan HOST_l2c(l,c) \
      (__builtin_write32_reversed (((unsigned int *)(c)),l),c+=4,l)

#definisikan SHA_LONG tanpa tanda int

#tentukan SHA_LBLOCK              16
#tentukan SHA_CBLOCK              (SHA_LBLOCK*4)
      /* SHA memperlakukan data masukan sebagai susunan bersebelahan nilai big-endian selebar 32 bit. */
#tentukan SHA_BLOK_TERAKHIR (SHA_CBLOCK-8)
#tentukan SHA_DIGEST_LENGTH 20

#tentukan SHA256_DIGEST_LENGTH            32

tipedef struct SHA256state_st {
            SHA_LONG h[8];
            SHA_LONG Nl,Nh;
            karakter tak bertanda data[SHA_CBLOCK]; int
            tak bertanda num;
            } SHA256_CTX;

#tentukan MD32_REG_T panjang
#tentukan PUTAR(a,n) (((a)<<(n))|((a)&0xffffffff)>>(32-(n))))

statis konstan SHA_LONG K256[64] =
            { 0x428a2f98UL,0x71374491UL, ... ... 0xbef9a3f7UL,0xc67178f2UL };

/* Spesifikasi FIPS merujuk pada rotasi ke kanan, sementara makro ROTATE kita merujuk ke kiri.
  * Inilah sebabnya Anda mungkin memperhatikan bahwa koefisien rotasi berbeda dari yang * diamati dalam dokumen
  FIPS oleh 32-N... */

#tentukan Sigma0(x) (PUTAR((x),30) ^ PUTAR((x),19) ^ PUTAR((x),10)) #tentukan Sigma1(x) (PUTAR((x),26)
^ PUTAR((x),21) ^ PUTAR((x),7)) #tentukan sigma0(x) (PUTAR((x),25) ^ PUTAR((x),14) ^ ((x)>>3))
#tentukan sigma1(x) (PUTAR((x),15) ^ PUTAR((x),13) ^ ((x)>>10))


#definisikan Ch(x,y,z) (((x) & (y)) ^ ((~(x)) & (z))) #definisikan Maj(x,y,z) (((x)
& (y)) ^ ((x) & (z)) ^ ((y) & (z)))

batalkan sha256_block_data_order (SHA256_CTX *ctx, const void *masuk) {
    tidak bertanda MD32_REG_T a,b,c,d,e,f,g,h, s0,s1,T1,T2,t;
    SHA_PANJANG              Bahasa Indonesia: X[16],l,Ki;
```

```
    int i;
    const unsigned char *data=masuk;

    Bahasa Indonesia:a = ctx->h[0];b = ctx->h[1];c = ctx->h[2];d = ctx->h[3];e = ctx->h[4];f = ctx->h[5];g =
    ctx->h[6];h = ctx->h[7];

    untuk (i=0;i<16;i++) {
            HOST_c2l(data,l); X[i] = l;
            Ki=K256[i];
            T1 = l + h + Sigma1(e) + Ch(e,f,g) + Ki;
            T2 = Sigma0(a) + Maj(a,b,c); h = g; g = f; f =
            e; d = c; c = b; b = a;                          Bahasa Indonesia:
                                                             e = d + T1; a = T1 + T2;
    }

    untuk (;i<64;i++) { s0 =
            X[(i+1)&0x0f]; s0 = sigma0(s0); s1 = X[(i+14)&0x0f]; s1 =
            sigma1(s1);
            Bahasa Indonesia: T1
            = X[i&0xf];
            Rumus untuk T1 adalah s0 + s1 + t.
            Bahasa Indonesia: X[i&0xf] = T1;
            Ki=K256[i];
            T1 += h + Sigma1(e) + Ch(e,f,g) + Ki;
            T2 = Sigma0(a) + Maj(a,b,c); h = g; g = f; f =
            e; d = c; c = b; b = a;                          Bahasa Indonesia:
                                                             e = d + T1; a = T1 + T2;
    }

    t=ctx->h[0]; ctx->h[0]=t+a; t=ctx->h[1]; ctx-
    >h[1]=t+b; t=ctx->h[2]; ctx->h[2]=t+c; t=ctx-
    >h[3]; ctx->h[3]=t+d; t=ctx->h[4]; ctx-
    >h[4]=t+e; t=ctx->h[5]; ctx->h[5]=t+f; t=ctx-
    >h[6]; ctx->h[6]=t+g; t=ctx->h[7]; ctx-
    >h[7]=t+h; kembali;



}

Bahasa Indonesia:void SHA256_Init (SHA256_CTX
    *c) { c->h[0]=0x6a09e667UL; c->h[1]=0xbb67ae85UL; c-
    >h[2]=0x3c6ef372UL; c->h[3]=0xa54ff53aUL; c->h[4]=0x510e527fUL;
    c->h[5]=0x9b05688cUL; c->h[6]=0x1f83d9abUL; c->h[7]=0x5be0cd19UL;
    c->Nl=0; c->num=0; kembali;
                        c->Nh=0;


}

void SHA256_addlength(SHA256_CTX *c, ukuran_t len) {
    SHA_LONG l, cNl,cNh; cNl=c-
    >Nl; cNh=c->Nh; l=(cNl+
    (((SHA_LONG)len)<<3))&0xffffffffUL; jika (l < cNl) /* luapan */ {cNh
    ++;} cNh += (len>>29);
```

```
    c->Nl=l; c->Nh=cNh; kembali;

}

batalkan SHA256_Perbarui (SHA256_CTX
                          *c, const batalkan *data_, ukuran_t len) {
    const unsigned char *data=data_; unsigned char *p;
    ukuran_t n, fragmen;


    SHA256_addlength(c, len); n = c->num;
    p=c->data; jika (n !
    = 0) fragment =
    SHA_CBLOCK-n;          {
        jika (len >= fragment) { memcpy
        (p+n,data,fragment);
              sha256_block_data_order (c,p); data +=
              fragment; len -= fragment; memset
              (p,0,SHA_CBLOCK); /*
              biarkan tetap nol */

        }
        else
             { memcpy (p+n,data,len); c->num =
              n+(unsigned int)len; kembali;

        }
    }
    sementara (len >= SHA_CBLOCK)
            { sha256_urutan_data_blok (c,data); data +=
            SHA_CBLOCK; len -=
            SHA_CBLOCK;
    }
    c->num=len; jika
    (len != 0) { memcpy
            (p,data,len);
    }
    kembali;
}

void SHA256_Final (unsigned char *md, SHA256_CTX *c) { unsigned char *p = c->data;
    ukuran_t n = c->num;

    SHA_LONG cNl,cNh;

    p[n] = 0x80; /* selalu ada ruang untuk satu */ n++;


    jika (n > (SHA_CBLOCK-8)) { memset
                (p+n,0,SHA_CBLOCK-n); n=0;

                urutan_data_blok_sha256 (c,p); } memset

    (p+n,0,SHA_CBLOCK-8-n);

    p += SHA_CBLOCK-8; cNh=c-
    >Nh; (void)HOST_l2c(cNh,p);
```

```
    cNl=c->Nl; (void)HOST_l2c(cNl,p); p -=
    SHA_CBLOCK;
    urutan_data_blok_sha256 (c,p); c->num=0;
    memset
    (p,0,SHA_CBLOCK); {unsigned long
    ll; unsigned int xn; untuk

      (xn=0;xn<PANJANG_DIGEST_SHA256/4;xn++)
                { ll=(c)->h[xn]; HOST_l2c(ll,md);
    }
    kembali;
}

void SHA256(const unsigned char *d, ukuran_t n,
                        unsigned char *md) {
    SHA256_CTX c;
    SHA256_Init(&c);
    SHA256_Perbarui(&c,d,n);
    SHA256_Final(md,&c); kembali;

}
```

B. SPESIFIKASI

Definisi big-endian-integer (isi: Z ÿint) : int := Int.or (Int.shl (isi 0)
(Int.repr 24))
(Int.atau (Int.shl (isi 1) (Int.repr 16))
(Int.atau (Int.shl (isi 2) (Int.repr 8)) (isi 3))).

Definisi LBLOCKz : Z := 16. (ÿ panjang blok, dalam 32-bit int ÿ)
Definisi CBLOCKz : Z := 64. (ÿ panjang blok, dalam karakter ÿ)

Definisi s256state := (daftar val ÿ (val ÿ (val ÿ (daftar val ÿ val))))%type.
Definisi s256-h (s: s256state) := fst s.
Definisi s256-Nl (s: s256state) := fst (snd s).
Definisi s256-Nh (s: s256state) := fst (snd (snd s)).
Definisi s256-data (s: s256state) := fst (snd (snd (snd s))).
Definisi s256-num (s: s256state) := snd (snd (snd (snd s))).

Induktif s256abs := (ÿ Keadaan abstrak SHA-256 ÿ)
S256abs: ÿ (hash: daftar int) (ÿ kata-kata di-hash, sejauh ini ÿ) (data:
daftar Z), (ÿ byte dalam blok parsial yang belum di-hash ÿ) s256abs.

Definisi s256a-regs (a: s256abs) : daftar int := **cocokkan**
a **dengan** data hash S256abs ÿ hash-blocks init-registers hash **akhir.**

Definisi s256a-len (a: s256abs) : Z := **cocokkan**
a **dengan** data hash S256abs ÿ (Zlength hashed ÿ 4 + Zlength data) ÿ 8 end%Z.

```
Definisi hilo hi lo := (Int.unsigned hi ÿ Int.modulus + Int.unsigned lo)%Z.
Definisi isbyteZ (i: Z) := (0 <= i < 256)%Z.

Definisi s256- relate (a: s256abs) (r: s256state) : Prop := cocokkan a
        dengan data hash S256abs ÿ
              s256-h r = peta Vint (hash-blok init-register yang di-hash)
            ÿ (ÿ hai, ÿ lo, s256-Nh r = Vint hai ÿ s256-Nl r = Vint lo ÿ
                    (Panjang Z hash ÿ 4 + data panjang Z)ÿ8 = hilo hi lo)%Z ÿ s256-
            data r = peta Vint (peta Int.repr data) ÿ (Panjang Z data
            < CBLOCKz ÿ Untuk semua isbyteZ data) ÿ (LBLOCKz | Panjang
            Z hash) ÿ s256-num r = Vint (Int.repr
            (data panjang Z)) akhir.


Definisi init-s256abs : s256abs := S256abs nil nil.

Definisi sha-finish (a: s256abs) : list Z := cocokkan a
  dengan data hash S256abs ÿ SHA-256 (intlist-to-Zlist hash ++ data) akhir.

Definisi cVint (f: Z ÿint) (i: Z) := Vint (fi).

Definisi sha256- panjang (len: Z) (c: val) : mpred := EX lo:int, EX
      hi:int, !! (hilo hi lo = len)
          dan (bidang-di Tsh t-struct-
          SHA256state-st -Nl (Vint lo) c ÿ bidang-di Tsh t-struct-SHA256state-
            st -Nh (Vint hi) c).

Definisi sha256state- (a: s256abs) (c: val) : mpred := EX r:s256state, !!
      s256- mengaitkan ar && data-at Tsh t-struct-SHA256state-st r c.

Definisi tuints (vl: daftar int) := ZnthV tuint (peta Vint vl).
Definisi tuchars (vl: daftar int) := ZnthV tuchar (peta Vint vl).

Definisi data-blok (sh: share) (isi: daftar Z) :=
    !! Untuk semua konten isbyteZ &&
    array-at tuchar sh (tuchars (peta Int.repr konten))
                  0 (konten Zlength).

Definisi --builtin-read32- reversed-spec :=
  DEKLARASIKAN ---builtin-read32- reversed
    DENGAN p: val, sh: share, konten: Z ÿint PRE [ 1
    DARI tptr tuint ]
          PROP() LOKAL (`(eq p) (eval-id 1))
          SEP (`(array-at tuchar sh (konten cVint) 0 4 p))
    POST [ tuint ]
          lokal (`(eq (Vint (isi bilangan bulat big-endian))) retval) dan `(array-at
          tuchar sh (isi cVint) 0 4 p).

Definisi --builtin-write32- reversed-spec :=
  DEKLARASIKAN ---builtin-write32- terbalik
    DENGAN p: val, sh: share, isi: Z ÿint
```

```
  SEBELUMNYA [ 1 DARI tptr tuint, 2 DARI tuint ]
      PROP(sh yang dapat ditulis bersama)
      LOKAL (`(eq p) (id-evaluasi 1);
             `(eq (Vint(isi bilangan bulat big-endian))) (id-evaluasi 2))
      SEP (`(blok memori sh (Int.repr 4) p))
  POST [ tvoid ]
      `(array-at tuchar sh (konten cVint) 0 4 p).

Definisi memcpy-spec := (ÿ dihilangkan ÿ)
Definisi memset-spec := (ÿ elided ÿ)

Definisi K-vektor : environ ÿ mpred :=
    array-di tuint Tsh (tuint K) 0 (panjang Z K).

Definisi sha256-block-data-order-spec :=
  DEKLARASIKAN -sha256-urutan-data-blok
  DENGAN hash: daftar int, b: daftar int, ctx: val, data: val, sh: bagikan
  PRE [ -ctx DARI tptr t-struct-SHA256state-st, - dalam DARI tptr tvoid ]
     PROP(Panjang Z b = LBLOCKz; (LBLOCKz | Panjang Z hash))
     LOKAL (`(eq ctx) (eval-id -ctx); `(eq data) (eval-id - masuk))
     SEP (`(array-di tuint Tsh
              (tuints (hash-blok init-register hash)) 0 8 ctx); `(data-blok sh
        (intlist-ke-Zlist b) data); `K-vektor (eval-var
        -K256 (tarray tuint 64)))
  POST [ tvoid ]
      (`(array-at tuint Tsh
            (tuints (hash-blok init-register (hashed++b))) 0 8 ctx) ÿ `(data-blok
       sh (intlist-ke-Zlist b) data) ÿ `K-vektor (eval-var
       -K256 (tarray tuint 64))).

Definisi SHA256-addlength-spec :=
 DEKLARASIKAN -SHA256-addlength
 DENGAN len : Z, c: val, n: Z
 PRE [ -c DARI tptr t-struct-SHA256state-st , - len DARI tuint ]
    PROP ( 0 <= n+lenÿ8 < dua-p 64)
    LOKAL (`(eq len) (`Int.unsigned (`force-int (eval-id - len)));
                `(eq c) (id-eval -c))
    SEP (`(sha256- panjang nc))
 POST [ tvoid ]
   `(sha256- panjang (n+lenÿ8) c).

Definisi SHA256- Init-spec :=
  DEKLARASIKAN -SHA256-
   Init DENGAN
   c : val PRE [ -c DARI tptr t-struct-SHA256state-st ]
           PROP () LOKAL (`(eq c) (eval-id -c))
           SEP(`(data-di- Tsh t-struct-SHA256state-st c))
  POST [ tvoid ]
            (`(sha256state-init-s256abs c)).

Pembaruan induktif -abs: daftar Z ÿs256abs ÿs256abs ÿProp :=
 Pembaruan-abs:
```

```
    (ÿ msg blok hash oldfrag newfrag, Zlength oldfrag
          < CBLOCKz ÿ Zlength newfrag <
          CBLOCKz ÿ (LBLOCKz | Zlength hash) ÿ
        (LBLOCKz | Zlength blok) ÿ oldfrag++msg
        = blok intlist-ke-Zlist ++ newfrag ÿ update-
        abs msg (S256abs hash oldfrag)

                        (S256abs (hashed++blocks) newfrag)).

Definisi SHA256-Update-spec :=
  DEKLARASIKAN -SHA256-Pembaruan
   DENGAN a: s256abs, data: daftar Z, c: val, d: val, sh: bagikan, len: nat
   PRE [ -c DARI tptr t-struct-SHA256state-st, -data- DARI tptr tvoid, - len DARI tuint ]
          PROP ((len <= panjang data)%nat; (s256a-
                      len a + Z.of-nat len ÿ 8 < dua-p 64)%Z)
          LOKAL (`(eq c) (id-eval -c); `(eq d) (id-eval -data- ); `(eq (Z.dari-
                nat len))
                    (`Int.tidak ditandatangani (`force-int (id-eval - len))))
          SEP(`K-vektor (eval-var -K256 (tarray tuint 64)); `(sha256state-
              ac); `(blok data sh data d))
  POST [ tayang ]
          EX a':-
           PROP (perbarui-abs (data len pertama) a a') LOKAL ()
           SEP(`K-vektor (eval-var -K256 (tarray tuint 64)); `(sha256state-
               a' c); `(data-blok sh data d)).

Definisi SHA256-Final-spec :=
  DEKLARASIKAN -SHA256-Final
   DENGAN a: s256abs, md: val, c: val, shmd: bagikan, sh: bagikan
   PRE [ -md DARI tptr tuchar, -c DARI tptr t-struct-SHA256state-st ]
          PROP (shmd berbagi-tulis)
          LOKAL (`(eq md) (eval-id -md); `(eq c) (eval-id -c))
          SEP(`K-vektor (eval-var -K256 (tarray tuint 64)); `(sha256state-
              ac); `(blok-memori shmd
              (Int.repr 32) md))
  POST [ tayang ]
          PROP () LOKAL ()
          SEP(`K-vektor (eval-var -K256 (tarray tuint 64)); `(data-di- Tsh
              t-struct-SHA256state-st c); `(data-blok shmd (sha-
              finish a) md)).
```