Verification of a Cryptographic Primitive: SHA-256

ANDREW W. APPEL, Princeton University

A full formal machine-checked verification of a C program: the OpenSSL implementation of SHA-256. This is an interactive proof of functional correctness in the Coq proof assistant, using the Verifiable C program logic. Verifiable C is a separation logic for the C language, proved sound w.r.t. the operational semantics for C, connected to the CompCert verified optimizing C compiler.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Correctness proofs; E.3 [**Data Encryption**]: Standards; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]

General Terms: Verification

1. INTRODUCTION

> *[C]ryptography is hard to do right, and the only way to know if something was done right is to be able to examine it. . . . This argues very strongly for open source cryptographic algorithms. . . . [But] simply publishing the code does not automatically mean that people will examine it for security flaws.* *Bruce Schneier [1999]*
>
> *Be suspicious of commercial encryption software . . . [because of] back doors. . . . Try to use public-domain encryption that has to be compatible with other implementations. . . . "* *Bruce Schneier [2013]*

That is, use widely used, well examined open-source implementations of published, nonproprietary, widely used, well examined, standard algorithms—because "many eyes make all bugs shallow" works only if there are many eyes paying attention.

To this I add: use implementations that are *formally verified with machine-checked proofs* of functional correctness, of side-channel resistance, of information-flow properties. "Many eyes" are a fine thing, but sometimes it takes them a couple of years to notice the bugs [Bever 2014]. Verification can guarantee program properties in advance of widespread release.

In this paper I present a first step: a formal verification of the functional correctness of the SHA-256 implementation from the OpenSSL open-source distribution.

Formal verification is not necessarily a *substitute* for many-eyes assurance. For example, in this case, I present only the assurance of functional correctness (and its corollary, safety, including absence of buffer overruns). With respect to other properties such as timing side channels, I prove nothing; so it is comforting that this same C program has over a decade of widespread use and examination.

SHA-256, the Secure Hash Algorithm with 256-bit digests, is not an encryption algorithm, but it is used in encryption protocols. The methods I discuss in this paper can be applied to the same issues that appear in ciphers such as AES: interpretation of standards documents, big-endian protocols implemented on little-endian machines, odd corners of the C semantics, storing bytes and loading words, signed and unsigned arithmetic, extended precision arithmetic, trustworthiness of C compilers, use of machine-dependent special instructions to make things faster, correspondence of models to programs, assessing the trusted base of the verification tools.

 This material is based on research sponsored by the DARPA under agreement number FA8750-12-2-0293. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

This paper presents the following result: I have proved functional correctness of the OpenSSL implementation of SHA-256, with respect to a *functional specification:* a formalization of the FIPS 180-4 *Secure Hash Standard* [FIPS 2012]. The machine-checked proof is done using the *Verifiable C* program logic, in the Coq proof assistant. Verifiable C is proved sound with respect to the operational semantics of C, with a machine-checked proof in Coq. The C program can be compiled to x86 assembly language with the CompCert verified optimizing C compiler; that compiler is proved correct (in Coq) with respect to the same operational semantics of C and the semantics of x86 assembly language. Thus, by composition of machine-checked proofs with no gaps, the assembly-language program correctly implements the functional specification.

In addition, I implemented SHA-256 as a functional program in Coq and proved it equivalent to the functional specification. Coq can execute the functional program on real strings (only a million times slower than the C program), and gets the same answer as standard reference implementations.[1] This gives some extra confidence that no silly things are wrong with the functional spec.

Limitations. The implementation is from OpenSSL, with some macro expansion to instantiate it from generic SHA-2 to SHA-256. I factor assignment statements so that there is at most one memory operand per command, e.g., ctx$\rightarrow$h[0] += a; becomes t=ctx$\rightarrow$h[0]; ctx$\rightarrow$h[0]=t+a; see §10.

CompCert generates assembly language, not machine language; there is no correctness proof of the assembler or of the x86 processor hardware on which one might run the compiled program. The Coq proof assistant is widely used, and its kernel is believed to be a correct implementation of the Predicative Calculus of Inductive Constructions (CiC), which in turn is believed to be consistent.

A different kind of limitation is in the time and cost of doing the verification. SHA-256 was the "shakedown cruise" for the *Verifiable C* system. This "cruise" revealed many inefficiencies of Verifiable C's proof automation system: it is slow, it is a memory hog and it is difficult to use in places, and it is *incomplete:* some corners of the C language have inadequate automation support. But this incompleteness, or shakiness of proof automation, cannot compromise the end-to-end guarantee of machine-checked logical correctness: every proof step is checked by the Coq kernel.

Nonlimitations. The other way that my implementation differs from OpenSSL is that I used the x86's byte-swap instruction in connection with big-endian 4-byte load/store (since it is a little-endian machine). This illustrates a common practice when implementing cryptographic primitives on general-purpose microprocessors: use machine-dependent special instructions to gain performance. It is good that the program logic can reason about such instructions.

What about using `gcc` or LLVM to compile SHA-256? Fortunately, these compilers (`gcc`, LLVM, CompCert) agree quite well on the C semantics, so a verification of SHA-256 can still add assurance for users of other C compilers. In most of the rare places they disagree, CompCert is correct and the others are exhibiting a bug [Yang et al. 2012]; no bugs have ever been found in the phases of CompCert behind Verifiable C.[2]

[1] That's 0.25 seconds per block, versus 0.25 microseconds; fast enough for testing the specification. The Coq functional program is a million times slower because it simulates the logical theory of binary integers used in the specification! The functional spec is *even slower than that,* because its W function takes a factor of 4^{16} more time.

[2] That is, CompCert has a front-end phase from C to C light; Verifiable C plugs in *after* this phase, at C light. Yang et al. [2012] found a bug or two in that front-end phase at a time when that phase was not formally verified, but they could not find *any* bugs in any of the verified phases, the ones between C light and assembly language. Since then, Leroy has formally verified the C-to-Clight phase, but that doesn't matter for Verifiable C, because in effect we verify functional correctness of the C light program. Also, Yang

2. VERIFIED SOFTWARE TOOLCHAIN

The Verified Software Toolchain (VST) [Appel et al. 2014] contains the *Verifiable C* program logic for the C language, proved sound with respect to the operational semantics of CompCert C. The VST has proof automation tools for applying this program logic to C programs.

One style of formal verification of software proceeds by applying a *program logic* to a program. An example of a program logic is Hoare logic, which relates the program c to its specification (precondition P, postcondition Q) via the judgment $\{P\}c\{Q\}$. This *Hoare triple* can be proved by using the inference rules of the Hoare logic, such as the *sequential composition* rule:

$$\frac{\{P\}\, c_1\, \{Q\} \qquad \{Q\}\, c_2\, \{R\}}{\{P\}\, c_1; c_2\, \{R\}}$$

We prefer *sound* program logics or analysis algorithms, i.e., where there is a proof that whatever the program logic claims about your program is actually true when the program executes. The VST is proved sound by giving a semantic model of the Hoare judgment with respect to the full operational semantics of CompCert C, so that we can really say, *what you prove in Verifiable C is what you get when the source program executes.* CompCert is itself proved correct, so we can say, *what you get when the source program executes is the same when the compiled program executes.* Composing these three proofs together: the proof of a program, the soundness of Verifiable C, and the correctness of CompCert, we get: *the compiled program satisfies its specification.*

C programs are tricky to verify because one needs to keep track of many side conditions and restrictions: this variable *is* initialized here, that addition *does not* overflow, this $p < q$ compares pointers into *the same* object, that pointer *is not* dangling. The Verifiable C logic keeps track of every one of these; or rather, assists the user in keeping track of every one. We know that there are no missed assumptions, because of the soundness proof w.r.t. the C semantics; and we know the C semantics does not miss any, because of the CompCert correctness proof w.r.t. safe executability in assembly language.

Of course, there are easier ways to prove programs correct. One can write functional programs in languages (such as Gallina, ML, Haskell) with much cleaner proof theories than C, and then the proof effort is smaller by an order of magnitude. Whenever the performance of a high-level garbage-collected language is tolerable, this is the way to go. The vast amount of software that is today written in Perl, Python, Javascript might profitably be rewritten in functional languages with clean proof theories for effective verification. But cryptographic primitives are not written in these languages; if we want to verify a well established widely used open-source cryptographic implementation, we need tooling for C.

Synthesis instead of verification?. C was not designed with a simple proof theory in mind, so perhaps a simpler route to verified crypto would be to use *program synthesis* from a domain-specific specification language. One example is Cryptol [Erkok et al. 2009] which can generate either C or VHDL directly from a functional specification. In principle one could hope to prove the Cryptol synthesizer correct (though this has not been done) or validate the output (which might be easier than proving general-purpose C programs).

---

et al. [2012] found a *specification* bug in CompCert, regarding how it treated bit-fields. Although Leroy has since fixed that specification bug, this also does not matter: Verifiable C is immune to specification bugs in C, for reasons discussed in §8.

Unfortunately, synthesis languages sometimes have limited expressiveness. Cryptol has been used to synthesize the block-shuffle part of SHA from a functional spec—but only the block-shuffle (the function that OpenSSL calls sha256_block_data_order).[3] Using Verifiable C I have verified the entire implementation, including the padding, length computation, multi-block handling, incremental update of unaligned strings, and so on. The Cryptol synthesizers (which translate Cryptol to C or VHDL) can handle only fixed-size blocks, so cannot handle these parts (SHA256_Init, SHA256_Update, SHA256_Final, SHA256).

3. SPECIFICATION OF SHA-256

A program without a specification *cannot be incorrect,* it can only be surprising.[4] Typically one might prove a C program correct with respect to a *relational specification*. For example, a C implementation implementing lookup tables must satisfy this relation between program states and inputs/outputs: that if the most recent binding for x is $x \mapsto y$, then looking up x yields y.

Sometimes one does this in two stages: prove that the C program correctly implements a *functional specification* (an abstraction of the implementation), then prove that functional specification satisfies the relational specification. For example, a C implementation implementing lookup tables by balanced binary search trees might be proved correct with respect to a functional-spec of red-black trees. Then the functional red-black trees can (more easily) be proved to have the lookup-table property.

For cryptographic hashing, we[5] built a functional spec from the FIPS 180-4 standard [FIPS 2012]. The relational spec is, "implements a random function." Unfortunately, nobody in the world knows how to prove that SHA-256 implements a random function—even on paper—so I did not attempt a machine-checked proof of that (see §11).

The FIPS 180-4 SHS (Secure Hash Standard) mentions (in §3.2) 32-bit unsigned binary arithmetic with operators such as addition-modulo-32, exclusive-or, and shifting. We must give a model of 32-bit arithmetic in pure logic. Fortunately, Leroy has defined such an Integers module and proved many of its properties as part of the semantics of CompCert C [Leroy 2009]; we use this directly in the functional spec, which is otherwise entirely independent of the C language. We have: the type int; operations such as Int.add, Int.xor; injection $(\mathsf{Int.repr} : \mathbf{Z} \to \mathsf{int})$ from the mathematical integers to 32-bit integers, and projection $(\mathsf{Int.unsigned} : \mathsf{int} \to \mathbf{Z})$. We have "axioms" such as,

$$\frac{0 \le i < 2^{32}}{\mathsf{Int.unsigned}\ (\mathsf{Int.repr}\ i)\ =\ i}$$

but this is not an axiom of the underlying logic (CiC), it is a theorem proved[6] from the axioms of Coq using the constructive definitions of Int.unsigned and Int.repr.

SHS defines SHA-256 on a bitstring of any length, and explains how to pack these into big-endian 32-bit integers. OpenSSL's implementation permits any sequence of bytes, that is, multiples of 8 bits. We represent a sequence of byte values by a sequence of mathematical integers, and we can define the big-endian packing function as,

```
Definition Z_to_Int (a b c d : Z) : Int.int :=
  Int.or (Int.or (Int.or (Int.shl (Int.repr a) (Int.repr 24)) (Int.shl (Int.repr b) (Int.repr 16)))
```

[3] Aaron Tomb, Galois.com, personal communication, 13 January 2014.
[4] Paraphrase of J. J. Horning, 1982.
[5] Stephen Yi-Hsien Lin wrote a functional spec of SHA-256 in Coq, which I subsequently adapted and rewrote.
[6] Henceforth, "proved" can be understood to mean, "proved with a machine-checked proof in Coq."

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

Given a list nl of byte values (represented as mathematical integers, type Z), if the length of nl is a multiple of 4, it's simple to define the corresponding list of big-endian 32-bit integers:

```
Fixpoint Zlist_to_intlist(nl: list Z): list int :=
  match nl with h1::h2::h3::h4::t ⇒ Z_to_Int h1 h2 h3 h4 :: Zlist_to_intlist t
                | _⇒ nil
  end.
```

Coq uses Definition for a nonrecursive function (or value, or type), and Fixpoint for structurally recursive functions. The operator :: is list *cons*. The operations such as Int.shl (shift left) and Int.repr (the 32-bit representation of a mathematical integer) are given foundational meaning by Leroy's Int package, as explained above.

SHS defines several functions *Ch, Maj, ROTR, SHR,* $\Sigma_0^{\{256\}}$, $\Sigma_1^{\{256\}}$ $\sigma_0^{\{256\}}$, $\sigma_1^{\{256\}}$, for example,

$$\begin{aligned} \mathsf{Ch}(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\ \mathsf{Maj}(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \end{aligned}$$

Translating these into Coq is quite straightforward:

```
Definition Ch (x y z : int) : int := Int.xor (Int.and x y) (Int.and (Int.not x) z).
Definition Maj (x y z : int) : int := Int.xor (Int.xor(Int.and x z)(Int.and y z))(Int.and x y).
Definition Rotr b x : int := Int.ror x (Int.repr b).
Definition Shr b x : int := Int.shru x (Int.repr b).
Definition Sigma_0 (x : int) : int := Int.xor (Int.xor (Rotr 2 x) (Rotr 13 x)) (Rotr 22 x).
Definition Sigma_1 (x : int) : int := ...
Definition sigma_0 (x : int) : int := ...
Definition sigma_1 (x : int) : int := ...
```

The vector $K_0^{\{256\}} \ldots K_{63}^{\{256\}}$ is given as a series of 32-bit hexadecimal constants, as is the vector $H_0^{(0)} \ldots H_7^{(0)}$. In Coq we write them in decimal, and inject with Int.repr:

```
Definition K := map Int.repr [1116352408 , 1899447441, 3049323471, ..., 3329325298].

Definition initial_registers := Map Int.repr [1779033703, 3144134277, ..., 1541459225].
```

Given a message M of length ℓ bits, the SHS explains: append a 1 bit, then enough zero bits so the length-appended message will be a multiple of the block size, then a 64-bit representation of the length. Since we have a message M of length n bytes; we append a 128 byte (which already has 7 trailing zeros), then the appropraiate number of zero bytes. We big-endian convert this to 32-bit integers, then add two more 32-bit integers representing the high-order and low-order parts of the length-in-bits.

```
Definition generate_and_pad M :=
 let n := Zlength M in
 Zlist_to_intlist (M ++ [128%Z] ++ list_repeat (Z.to_nat (-(n + 9) mod 64)) 0)
                  ++ [Int.repr (n * 8 / Int.modulus), Int.repr (n * 8)].
```

Note that $0 \leq a \bmod 64 < 64$ even if a is negative. The magic number 9 comes from 1+8: 1 terminator byte (value 128) plus 8 bytes for the 64-bit length field. Taking $-(n+9)$ mod 64 gives the number of bytes of padding necessary to round up to the next multiple of 64 bytes, which is the block size.

SHS defines the message schedule W_t as follows:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

where the superscript (i) indicates the value in the ith message block. We translate this into Coq as,

```
Function W (M: Z → int) (t: Z) {measure Z.to_nat t} : int :=
  if zlt t 16
  then M t
  else (Int.add (Int.add (sigma_1 (W M (t-2))) (W M (t-7)))
         (Int.add (sigma_0 (W M (t-15))) (W M (t-16)))).
Proof.
intros; apply Z2Nat.inj_lt; omega. (* t-2 < t *)
intros; apply Z2Nat.inj_lt; omega. (* t-7 < t *)
intros; apply Z2Nat.inj_lt; omega. (* t-15 < t *)
intros; apply Z2Nat.inj_lt; omega. (* t-16 < t *)
Qed.
```

Coq is a language of total functions. The measure and Proof/Qed demonstrate that the W function always terminates. There is one proof line for each of the 4 recursive calls; each proof is, "calling on a smaller value of t."

One could run this W as a functional program; but it takes time exponential in t, since there are 4 recursive calls. It serves well as a functional spec but it is not practically executable.

The block cipher computes 256-bit (8-word) hashes of 512-bit (16-word) blocks. The accumulated hash of the first i blocks is the vector $H_0^{(i)} \ldots H_7^{(i)}$. To hash the next block, the eight "working variables" a through h are initialized from the $H^{(i)}$ vector. Then 64 iterations of this `Round` function are executed:

```
Definition registers := list int.

Function Round (regs: registers) (M: Z → int) (t: Z)
      {measure (fun t ⇒ Z.to_nat(t+1)) t} : registers :=
 if zlt t 0 then regs
 else match Round regs M (t-1) with
  | [a,b,c,d,e,f,g,h] ⇒
    let T1 := Int.add(Int.add(Int.add(Int.add h (Sigma_1 e))(Ch e f g))(nthi K t))(W M t) in
    let T2 := Int.add (Sigma_0 a) (Maj a b c) in
     [Int.add T1 T2, a, b, c, Int.add d T1, e, f, g]
  | _⇒ nil
  end.
Proof. intros; apply Z2Nat.inj_lt; omega. Qed.
```

That is, one calls (Round r (nthi block) 63). (The function `nthi` b i returns the ith element of the list b, or the arbitrary element Int.zero if i is negative or beyond the length of the list.) If the length of the regs list is not 8, an arbitrary result (the empty list) is returned; but it *will* be 8.

I represent registers as a list, rather than a dependently typed vector (i.e., a list whose type inherently enforces the length restriction) to keep the specification as first-order as possible. This simplifies reasoning about the specification, especially its portability to other logics.

The round function returns registers $a', b', \ldots, h'$ which are then added to the $H^{(i)}$ to yield $H^{(i+1)}$:

```
Definition hash_block (r: registers) (block: list int) : registers :=
      map2 Int.add r (Round r (nthi block) 63).
```

Given a message of length $16k$, the following function computes the $H^{(k)}$ by applying hash_block to each successive 16-word block:

```
Function hash_blocks (r: registers) (msg: list int) {measure length msg} : registers :=
  match msg with
  | nil ⇒ r
  | _⇒ hash_blocks (hash_block r (firstn 16 msg)) (skipn 16 msg)
  end.
Proof. ... Qed.
```

Finally, the SHA_256 produces the message digest as a 32-byte string by the big-endian conversion of $H^{(k)}$.

```
Definition SHA_256 (str : list Z) : list Z :=
    intlist_to_Zlist (hash_blocks init_registers (generate_and_pad str)).
```

4. FUNCTIONAL PROGRAM

One can prove that a program satisfies a specification, but how does one know that the specification is properly written down? One way to gain confidence in the specification is to calculate its results on a series of examples, i.e., to run it. The SHA_256 function given above is actually an *executable specification*. Coq permits relational (nonconstructive, propositional) specifications that do not run, but also permits fully constructive specifications such as this one.

However, since the W function is exponential, it's impractical to run this program. Therefore I wrote an alternative functional program called SHA_256'. One can run this program directly inside the Coq proof assistant on actual inputs; it takes about 0.25 seconds per block.

The key to this "efficiency" is that the W function should remember its previous results.[7] Here msg is the *reversed* list $W_{t-1}, W_{t-2}, W_{t-3}, \ldots W_0$:

```
Definition Wnext (msg : list int) : int :=
 match msg with
 | x1::x2::x3::x4::x5::x6::x7::x8::x9::x10::x11::x12::x13::x14::x15::x16::_ ⇒
      (Int.add (Int.add (sigma_1 x2) x7) (Int.add (sigma_0 x15) x16))
 | _ ⇒ Int.zero (* impossible *)
 end.
```

Should we be worried about the "impossible" case? Coq is a language of total functions, so we must return *something* here. One reason we need not worry is that I *proved* that this case cannot cause the SHA_256' program to be wrong. That is, I proved the equivalence (using the extensionality axiom):

```
Lemma SHA_256'_eq: SHA_256' = SHA_256.
```

Also, the fact that SHA_256' gives the right answer—on all the inputs that I tried—allows us to *know* that SHA_256 is also right on those inputs.

[7] Also in this efficient program the generate_and_pad function is done quite differently.

This equivalence proof took about a day to build; this is much faster than building the proof that the C implementation correctly implements the functional spec. But sometimes we must program in C—to get SHA that runs in microseconds rather than seconds.

Instead of calculating the result inside Coq, one could instead extract the program as an ML program, and compile with the OCaml compiler. This would lead to faster results than Coq, but slower than C. For the purpose of testing the SHA specification, it is unnecessary.

5. INTRODUCTION TO VERIFIABLE C

The *Verifiable C* language and program logic is a subset of CompCert's *C Light* language. Every Verifiable C program is a legal C program, and every C program can be expressed in Verifiable C with only local program transformations, such as pulling side effects out of expressions: a=(b+=2)+3; becomes b+=2; a=b+3; (sometimes an extra local variable is required to hold the intermediate result). The CompCert compiler accepts (essentially) the full C language; we use a subset not because of any inadequacy of CompCert but to accommodate reasoning about the program. It is easier to reason about one assignment at a time.

Separation logic. We use a variant of Hoare logic known as *separation logic*, which is more expressive regarding anti-aliasing of pointers and separation of data structures. We write $\{P\}\, c\, \{Q\}$ to mean (more or less), if P holds before c executes, then Q will hold after. In our separation logic, the assertion P has a *spatial* part dealing with the contents of *a particular footprint* of memory, and a *local* part dealing with local program variables, and a *propositional* part dealing with mathematical variables. An example of a *spatial* assertion is,

$$\mathsf{array}^{f}_{[0,n)}(p) \qquad\qquad (n : \mathbf{Z},\ f : \mathbf{Z} \to \mathbf{V},\ p : \mathbf{V})$$

which represents an array of n elements starting at address p, whose ith element is $f(i)$. Here f is a total function from (mathematical) integers to values; we ignore f's domain outside $[0, n)$. Values $\mathbf{V}$ may be 32-bit integers (Vint i), 32-bit representations of mathematical integers (Vint (Int.repr z)), floating point (Vfloat f), pointers (Vptr b i) with base b and in-the-block offset i, or undefined/uninitialized values (Vundef).

Verifiable C's array constructor actually takes two more arguments: a permission share π indicating read-only, read-write, etc.; and the C-language type of the elements, such as the type of unsigned characters,

Definition tuchar := Tint I8 Unsigned noattr.

Suppose we have two different arrays p, q and we execute the assignment p[i]=q[j]; one possible specification is this:

$$\begin{array}{l}\{0 \le i < j < n \ \wedge\ (\mathsf{array}^{f}_{[0,n)}(p)\ *\ \mathsf{array}^{g}_{[0,n)}(q))\}\\ \mathtt{t} = \mathtt{q[j]};\quad \mathtt{p[i]} = \mathtt{t};\\ \{0 \le i < j < n \ \wedge\ (\mathsf{array}^{f[i:=g(j)]}_{[0,n)}(p)\ *\ \mathsf{array}^{g}_{[0,n)}(q))\}\end{array}$$

Separation logic's inference rules prefer reasoning about one load or store at a time, so I have made a local program transformation. Here I assume there are two disjoint arrays p and q whose contents are f and g respectively. You can tell they are disjoint because the $*$ operator enforces this. Because they are disjoint, we know (in the post-condition) that q is unchanged, i.e., its contents are still g. (If the programmer had intended p and q to possibly overlap, one would write a different specification.)

Program variables, symbolic values. This is a bit of a simplification: i, j, p, q are program variables, not logical variables. Verifiable C distinguishes these; one might write the precondition "for real" as,

PROP ($0 \le i < j < n$; writable_share π_1)
LOCAL(`(eq $i$) (eval_id _i); `(eq j) (eval_id _j); `(eq $p$) (eval_id _p); `(eq q) (eval_id _q))
SEP (`(array_at tuchar $\pi_1$ $f$ 0 $n$ $p$); `(array_at tuchar π_2 g 0 n q))

where the PROP part has pure logical propositions (that do not refer to program state); LOCAL gives assertions about local variables of the program state (but not memory); and SEP is the *sep*arating conjunction of *spatial* assertions, i.e., about various disjoint parts of memory.

The notation `(eq $i$) (eval_id _i) means, "C program variable _i contains the symbolic value $i$. This is effectively a statement about the current program state's local-variable environment $\rho$. The notation `f lifts f over local-variable environments [Appel et al. 2014, Chapter 21], that is,

`(eq i) (eval_id _i) = (fun $\rho \Rightarrow$ (eq i) (eval_id _i ρ)) =
(fun $\rho \Rightarrow$ (fun $x \Rightarrow i = x$) (eval_id _i ρ)) = (fun $\rho \Rightarrow i$ = eval_id _i ρ).

or in other words, looking up _i in ρ yields i.

Permissions. The p array needs to be writable, while the q array needs to be at least read-only. This is expressed with permission-shares: p's permission-share π_1 needs to satisfy the writable_share predicate. We don't need to say readable_share π_2 because that is implied by the array_at predicate.

We call these *permission shares* rather than just *permissions* because in the shared-memory concurrent setting, a proof could split $\pi_1 = \pi_{1a} + \pi_{1b}$ into smaller shares that are given to concurrent threads. These shares π_{1a} and π_{1b} would not be strong enough for write permission, but they could be both strong enough for read permission. That permits exclusive-write-concurrent-read protocols. Now, suppose SHA-256 were called in one thread of a concurrent program. Its parameter (the string to be hashed) could be a read-only shared array, but its result (the array to hold the message digest) must be writable. All this is concisely expressed in the permission-share annotation of my SHA-256 specification.

Control flow. The C language has control flow: a command c might fall-through normally, might `continue` a loop, or `break` a loop, or `return` from a function. Thus the postcondition Q must have up to four different assertions for these cases. For the case where all but fall-through are prohibited—i.e., three of these four postcondition-assertions are **False**—use the construction normal_ret_assert.

normal_ret_assert (
 PROP ()
 LOCAL(`(eq $i$) (eval_id _i); `(eq j) (eval_id _j); `(eq $p$) (eval_id _p); `(eq q) (eval_id _q))
 SEP (`(array_at tuchar $\pi_1$ (upd $f$ $i$ ($q$ $j$)) 0 $n$ $p$); `(array_at tuchar π_2 g 0 n q)))

This postcondition shows that the p array has changed in one spot and q has not changed. We can omit $(0 \le i < j < n)$ from the postcondition, since it's a logical fact independent of state, and (if true in the precondition) is eternally true.

Higher-order reasoning. Ordinary separation logic is inexpressive regarding function-pointers, data abstraction, and concurrency; so Verifiable C is a higher-order impredicative concurrent separation logic. Higher-order means that one can quantify over predicates. This is useful for specifying abstract data types. It is also useful for function pointers: if function f takes a parameter p that's a function-pointer, then the

precondition of f will characterize the *specification* of p, i.e., p's precondition and postcondition. When function-pointer specifications are used to describe object-oriented programs, then impredicative quantification over these specifications is necessary.

One might think that C is not an object-oriented language, but in fact C programmers often use design patterns that they express with void $*$. C's type system is to weak to "prove" that all these void $*$ casts turn out all right, but we can specify and prove this with the program logic.

The SHA verification does not use these higher-order features, though one could use data abstraction for the context structure, SHA256state_st. However, OpenSSL uses an object-oriented "engine" construction to compose HMAC with SHA.

Further reading. Appel et al. [2014] give a full explanation of the program logic.

6. THE C PROGRAM

The OpenSSL implementation of SHA-256 is clever in several ways (many of which were intentional in the SHA-256 design):

(1) It works in one pass, waiting until the end before adding the padding and length.
(2) It allows incremental hashing. Suppose the message to be hashed is available, sequentially, in segments $s_1, s_2, \ldots, s_j$. One calls SHA256_Init to initialize a context, SHA256_Update with each s_i in turn, then SHA256_Final to add the padding and length and hash the last block. If the s_i are not block-aligned, then SHA256_Update remembers partial blocks in a buffer. However, a block internal to one of the s_i is not cycled through the buffer; the sha256_block_data_order function operates on it directly from the memory where it was passed to SHA256_Update.
(3) Within the 64-round computation, it store *only* the the most recent 16 elements of W_t, in a buffer accessed modulo 16 using bitwise-and in the array subscript.
(4) In adding the length of s_i to the accumulated 64-bit count of bits, there is an overflow test: is the result of $(a + b) \mod 2^{32} < a$? If so, add a carry to the high-order word. Such tests are easy to get wrong [Wang et al. 2013]; here it works because a, b are declared unsigned, but still a proof is worthwhile.
(5) In the SHA256_Final function, there is one last block containing the 1-bit, padding, and length. But there could be two "last" blocks, if the message body ends within 8 bytes of the end of a block (so there's no room for the 1-bit plus 64-bit length).
(6) The accumulated state between calls to SHA256_Update is kept in a record "owned" by the caller and initialized by SHA256_Init. But the W vector is purely local to the "round" function (sha256_block_data_order), so is kept as a local-variable (stack-allocated) array. Although that's not *particularly* clever, it's too clever for some C-language verification systems, which (unlike Verifiable C) cannot handle addressable local variables [Greenaway et al. 2012; Carbonneaux et al. 2014].

The client of SHA-256 calls upon it as follows:

```
typedef struct SHA256state_st {
    unsigned int h[8];      // The H vector
    unsigned int Nl,Nh;     // Length, a 64-bit number in two parts
    unsigned char data[64]; // Partial block not yet hashed
    unsigned int num;       // Length of the message fragment
} SHA256_CTX;

SHA256_CTX c;
char digest[32];
char *m_1, *m_2, ..., *m_k;
unsigned int n_1, n_2, ..., n_k;
```

// How the caller hashes a message:
SHA256_Init(&c);
SHA256_Update(&c, m_1, n_1);
SHA256_Update(&c, m_2, n_2);
...
SHA256_Update(&c, m_k, n_k);
SHA256_Final($digest$, &c);

The strings m_i of lengths n_i respectively make up the message. The idea is that Init sets up the context c with the initial register state c.h[ ], and then each Update hashes some more blocks into that register state. If m_i is not a full block, or rather if $\sum_{j=1}^{i} n_j$ is not a multiple of the block size, then a partial block is saved in (copied into) the context c. Then the $(i+1)$th call to Update will use that fragment as the beginning of the next full block. The m_i need not be disjoint; the caller can build the successive parts of the message in the same m buffer.

After the ith call, the registers c.h[ ] contain the hash of all the full blocks seen so far, and the length Nl,Nh contains the length (in bits) of all the message fragments, i.e., $8 \cdot \sum_{j=1}^{i} n_j$.

At the end, the Final call adds the padding and length, and hashes the last block(s). The final c.h[ ] values are then returned as a byte-string, the message $digest$.

Most of these "clever" implementation choices are not directly visible in the functional specification, and are not representable in a domain-specific language such as Cryptol. They are just general-purpose C programming, and our specification language must be able to reason about them.

7. SPECIFYING THE C PROGRAM

The Separation Logic specification of a C program relates the program (and its in-memory data structures) to functional or relational correctness properties. Appendix B gives the full separation-logic specification of the OpenSSL SHA-256 program; here I present a part of it.

The SHA256_CTX data structure has a *concrete* meaning and an *abstract* meaning. The concrete meaning is given by this 6-tuple of values, corresponding to the 6 fields of the struct:

```
Definition s256state := (list val * (val * (val * (list val * val)))).
                (*comment:  h      Nl     Nh     data   num *)
```

There's a specific reason for using a tuple here, instead of a Coq record: this tuple type is calculated automatically from the C-language *struct* definition, *inside* Coq's calculational logic.

The abstract meaning is that all the full blocks of $m_1 + m_2 + \ldots + m_i$ have been parsed[8] into a sequence of 32-bit words that we call hashed; and the remaining less-than-a-block fragment is a sequence of bytes that we call data.

[8]SHS uses the word "parsed" to indicate: grouping bytes/bits into big-endian 32-bit words, and grouping 32-bit words into 16-word blocks.

```
Inductive s256abs :=      (* SHA-256 abstract state *)
 S256abs: ∀ (hashed: list int)   (* words hashed, so far *)
              (data: list Z),    (* bytes in partial block *)
               s256abs.
```

This fancy notation is really just a 2-tuple (hashed,data); I define it this way to influence the names Coq chooses for introduced variables.

The abstract state is an abstraction of the concrete state. I make this relation formal in Coq as follows. First, we calculate what the H vector would be at the end of hashed:

```
Definition s256a_regs (a: s256abs) : list int :=
 match a with S256abs hashed data ⇒ hash_blocks init_registers hashed end.
```

Notice that this calls upon hash_blocks from the functional spec described in section 3.

Next, we can calculate the bit-length of the hashed words plus the data bytes:

```
Definition s256a_len (a: s256abs) : Z :=
  match a with S256abs hashed data ⇒ (Zlength hashed * 4 + Zlength data) * 8 end.
```

We can define the 64-bit concatenation of two 32-bit numbers, and what it means for a (mathematical) integer to be representable in an unsigned char:

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

Finally, here is the abstraction relation:

```
Definition s256_relate (a: s256abs) (r: s256state) : Prop :=
  match a with S256abs hashed data ⇒
    s256_h r = map Vint (hash_blocks init_registers hashed)
  ∧ (∃ hi, ∃ lo, s256_Nh r = Vint hi ∧ s256_Nl r = Vint lo ∧
        (Zlength hashed * 4 + Zlength data)*8 = hilo hi lo)
  ∧ s256_data r = map Vint (map Int.repr data)
  ∧ (length data < 64 ∧ Forall isbyteZ data)
  ∧ (16 | Zlength hashed)
  ∧ s256_num r = Vint (Int.repr (Zlength data))
  end.
```

That is, a concrete state $(r_{\rm h}, r_{\rm Nh}, r_{\rm Nl}, r_{\rm data}, r_{\rm num})$ represents an abstract state $a = (hashed, data)$ whenever:

— $r_{\rm h}$ is the result of hashing all of $hashed$;
— the bit-length of $(hashed, data)$ equals $r_{\rm Nh} \cdot 2^{32} + r_{\rm Nl}$;
— the sequence of char values $r_{\rm data}$ corresponds exactly to the sequence of (mathematical) integers $data$;
— the length of $data$ is less than the block size, and every element of data is $0 \le d < 256$;
— the length of $hashed$ is a multiple of 16 words;
— the length of $data$ is $r_{\rm num}$ bytes.

Verifiable C's logic has an operator (data_at π τ r p) saying that memory-address p, interpreted according to the C-language type τ, contains (struct/array/integer) data value r with access permission π. For example: τ =t_struct_SHA256state_st, p is a pointer to struct SHA256state_st, r is a concrete-state value $(r_{\rm h}, r_{\rm Nh}, r_{\rm Nl}, r_{\rm data}, r_{\rm num})$, and π is the full-access permission Tsh.

To relate the in-memory SHA256_CTX to an abstract state, we simply compose the relations data_at and s256_relate:

Definition sha256state_ (a: s256abs) (c: val) : mpred :=
EX r: s256state,
PROP (s256_relate a r)
LOCAL ()
SEP (data_at Tsh t_struct_SHA256state_st r c).

This relates a to c by saying there exists a concrete state r such that abstract-to-concrete composes with concrete-in-memory.

Incremental update. The SHA256_Update function updates a context c with the bytes data_ of length len:

```
void SHA256_Update (SHA256_CTX *c, const void *data_, size_t len);
```

Suppose data_ contains the sequence of integers msg. Appending msg to an abstract state $a = (hashed, oldfrag)$ yields the updated abstract state $a' = (hashed{+}{+}blocks, newfrag)$ when,

Inductive update_abs: list Z → s256abs → s256abs → Prop :=
Update_abs:
(∀ $msg\ hashed\ blocks\ oldfrag\ newfrag$,
Zlength $oldfrag < 64 \rightarrow$
Zlength $newfrag < 64 \rightarrow$
(16 | Zlength $hashed$) →
(16 | Zlength $blocks$) →
$oldfrag{+}{+}msg$ = intlist_to_Zlist $blocks$ ++ $newfrag \rightarrow$
update_abs msg (S256abs $hashed\ oldfrag$) (S256abs ($hashed{+}{+}blocks$) $newfrag$)).

where intlist_to_Zlist unpacks big-endian 32-bit words into a sequence of byte values.

With these preliminaries defined, I can now present the separation-logic specification of the Update function.

Definition SHA256_Update_spec :=
DECLARE _SHA256_Update
WITH a: s256abs, $data$: list Z, c : val, d: val, sh: share, len : nat
PRE [_c OF tptr t_struct_SHA256state_st, _data_ OF tptr tvoid, _len OF tuint]
PROP (len <= length $data$;
s256a_len a + Z.**of**_nat $len * 8 <$ two_p 64)
LOCAL (`(eq $c$) (eval_id _c); `(eq d) (eval_id _data_);
`(eq (Z.**of**_nat $len$)) (`Int.unsigned(`force_int(eval_id _len))))
SEP(`K_vector (eval_var _K256 (tarray tuint 64));
`(sha256state_ $a$ $c$); `(data_block sh $data$ d))
POST [tvoid]
EX a':s256abs,
PROP (update_abs (firstn len $data$) a a') LOCAL ()
SEP(`K_vector (eval_var _K256 (tarray tuint 64));
`(sha256state_ $a'$ $c$); `(data_block sh $data$ d)).

DECLARE gives the name (C language function identifier) of the function being specified. WITH binds (logical/mathematical) variables that can be used in both the precondition and postcondition.

The precondition has the form PRE [$\vec{x}$] PROP P LOCAL Q SEP R where $\vec{x}$ are the function parameters, annotated with their C language types (e.g., data_ has type pointer-to-void); P are pure logical PROPositions (that do not refer to the input state); Q are local facts (that do not refer to the memory, but may refer to program variables), and R are spatial facts (that refer to the memory via SEParation logic).

Here, $data$ is a sequence of integers and d is an address in memory; both of these are logical variables. The LOCAL clause says that the function's data_ parameter actually contains the value d, the c parameter contains the pointer value c, and so on. The SEP clause names three *separate* memory regions of interest: the 64-word global array $K256$; a SHA256_CTX c representing abstract state a; and a data-block at address d containing the next message-segment $data$.

The postcondition is parameterized by the return value (this particular function has no return value). Its PROP part relates a to the new abstract state a'; the LOCAL part is empty (since there's no return value to characterize), and the spatial (SEP) part says: the global array K256 is still there unchanged; at address c there is now an updated state a'; and the data-block at d is still there unchanged.

Corollary. By the nature of separation logic, this functional specification inherently makes specific guarantees about confidentiality, integrity, and lack of buffer overruns:

— The only variables or data structures read or written to are those mentioned in function's specification.
— The only variables or data structures written to are those mentioned with write permission in the function's specification.
— The values written are limited to what the specification claims.

For example, SHA256_Update_spec's SEP clauses mention only the memory blocks at addresses _K256, c, and d; and the permission-share sh (controlling d) is not mentioned as writable; this severely limits where SHA256_Update can read and write.

Static analysis tools such as Coverity cannot prove functional correctness, but at least in principle they can find basic safety problems. But "Coverity does not spot the heartbleed flaw ... it remained stubborn even when they tweaked various analysis settings. Basically, the control flow and data flow between the socket read() from which the bad data originates and the eventual bad memcpy() is just too complicated." [Regehr 2014]

SHA-256, especially the Update function which copies fragments of arrays, contains nontrivial control flow and data flow leading to memcpy(). But the full verification reported in this paper *must* fully analyze the control and data, no matter how complicated; and the higher-order logic (CiC) provides a sufficiently expressive tool in which to do it. We know there's no heartbleed in SHA.

8. IS THE SPECIFICATION RIGHT?

The SHA-256 program is about 235 lines of C code (including blank lines and sparse comments).

The FIPS 180-4 specification of SHA is 35 pages of text and mathematics; the parts specific to SHA-256 are perhaps 16 pages. My functional specification—the "translation" of this to logic—is 169 lines of Coq, but it relies on libraries for the mathematical theories of the unbounded integers, the 32-bit integers, and lists, which together are many lines more.

My specification of how the C code corresponds function-for-function to the functional spec takes 247 lines of Coq. The proof in Coq is much larger—see §9—but it need not be trusted because it is machine-checked.

Have we gained anything when the specification of a 235-line program is 169+247 lines? Is it easier to understand (and trust) the program, or its specification?

There are several reasons that the specification is valuable:

(1) It can be manipulated logically. For example, two different characterizations of the W function can be proved equivalent.

(2) The functional specification can be executed on test data. In this case, we do this indirectly but assuredly by proving its equivalence to a functional program that executes on the test data.
(3) The proof of SHA-256 correctness connects directly to the proof of C-compiler correctness, *inside the theorem prover with no specification "gaps."*

The last point is quite important. The C program for SHA-256 may be only 235 lines, but its meaning depends on the understanding of the semantics of C. Even if it is true that (these days) C has a clear and well-understood specification,[9] that spec is orders of magnitude larger than 235 lines. In contrast, in the Verified Software Toolchain the C spec is an *internal interface* between the program logic and the CompCert compiler. That is, suppose for the sake of argument that CompCert's C specification is "wrong;" or the Verifiable C program logic is "wrong." It still won't matter: by composing the correctness proof of SHA-256 (in the Verifiable C program logic), with the soundness proof of the program logic, with the correctness proof of CompCert, we get an end-to-end proof about the observable input/output behavior of the assembly language program, regardless of the internal specifications.

Still, the first point is important too: the value of a specification is that one can interact with it in logic. You need not assume that my translation of the SHS document into Coq is correct; you have the opportunity to test its properties mathematically (by proving theorems about it) in the proof assistant.

The specifications of the block_data_order, Init, Update, and Final functions are rather complex, because of the way they support incremental hashing. One mathematical way of gaining confidence in these specifications is to compose them. That is, this C function

```
void SHA256(const unsigned char *d, size_t n, unsigned char *md) {
    SHA256_CTX c;
    SHA256_Init(&c);
    SHA256_Update(&c,d,n);
    SHA256_Final(md,&c);
}
```

should be equivalent to nonincremental SHA-256 hashing. We can see this in its spec:

```
Definition 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 (writable_share msh;     Z.of_nat (length data) * 8 < two_p 64)
   LOCAL (`(eq d) (eval_id _d); `(eq (Z.of_nat (length data))) (`Int.unsigned (`force_int (eval_id _n)));
          `(eq md) (eval_id _md))
   SEP(`K_vector (eval_var _K256 (tarray tuint 64));
       `(data_block dsh data d); `(memory_block msh (Int.repr 32) md))
  POST [ tvoid ]
   SEP(`K_vector (eval_var _K256 (tarray tuint 64));
       `(data_block dsh data d); `(data_block msh (SHA_256 data) md)).
```

This says that calling SHA256(d,n,md) will fill in the message-digest md with the hash as computed by the functional specification (SHA_256 $data$), as long as the memory at d was indeed $data$, and $data$ is less than two billion gigabytes. In addition, the (global) K256 must be properly initialized beforehand, and is guaranteed preserved

[9] and even if it were true that compiler experts understand that specification *in the same way* that programmers do, which is doubtful [Wang et al. 2013]

unchanged; the input data must be present and will not be modified; and the output area must be writable. Finally, no other memory (except for the activation records of called functions) will be read or written; this is an implicit (but very real) guarantee of the separation logic.

The fact that SHA256 satisfies this (relatively) simple specification is a *proof* for the nonincremental case, that all the other functions' specifications are "right"—that is, they compose properly. It is not a proof that the incremental case (more than one call to Update) is specified right, but it does help build assurance. Guarantees about the incremental case rely on the rightness of the SHA256_Update_spec definition.

9. THE PROOF

The proof of the functional program w.r.t. the functional specification is fairly concise:

Lines	Seconds	component
1022	29	Lemmas about the functional spec
1202	12	Correctness proof of functional program
2424	41	*Total*

I show here the size of the Coq proof, and the time for Coq to check the proof in batch mode. The first component (lemmas) is shared with the proof of the C program.

My correctness proof of the C program is quite large—6539 lines of proof, written by hand with some cut-and-paste. It's also very slow to check. Thus, the current Verifiable C system must be regarded as a prototype implementation (though unlike most prototypes it has a machine-checked proof of correctness).

Lines	Seconds	component
1022	29	Lemmas about the functional spec
229	83	Proof of addlength function
1640	625	sha256_block_data_order()
43	256	SHA256_Init()
1682	800	SHA256_Update()
1484	687	SHA256_Final()
58	91	SHA256()
6539	2571	*Total*

Writing 6500 lines to verify this program is simply too much work. I'm sure that my proof is clumsy and inelegant in many places and likely there is a 2000-line proof struggling to get out. But perhaps the real solution here is to drastically improve the proof automation by using modern proof-search algorithms such as satisfiability modulo theories (SMT). Recent experiments have combined the trustworthiness of Coq (small kernel checking a proof) with the power of SMT (large C++ program claiming unsatisfiability) by exporting proof witnesses from SMT solvers to Coq [Besson et al. 2011; Armand et al. 2011].

Checking the proofs takes 2571 seconds (43 minutes) on one processor of an Intel core i7 (at 3.4 Ghz) with plenty of cache and 2GB ram. Multicore, it goes much faster using parallel make. Still, 43 minutes is far too long for a program this size. As a batch command it might be tolerable; the problem is in the interactive proof, where it might take 2 minutes to move past one line of C code—this is not very interactive.

Coq is not normally so slow; the problem is symbolic execution. The component listed in the first line of the table above (lemmas about the functional spec etc.) contains no symbolic execution—no application of the C-language program logic with all its side conditions. That component takes only 29 seconds in Coq.

Why is symbolic execution of C programs so slow? We wrote the prototype interactive prover for Verifiable C as a user-driven symbolic executor programmed in the Ltac language of Coq. As symbolic execution proceeds, the user frequently provides proofs for those steps where the automation cannot find a proof. So far, no problem—although it would be better to have more automation, so less user interaction; that's future work. The problem is that Coq builds a proof trace of the symbolic execution—that is, every step of the analysis corresponds to a data structure describing a proof term. Worse yet, as the terms are being constructed they are actually function closures (activation records) that (when invoked) will build the concrete proof terms. Since each step of symbolic execution checks dozens of conditions, the proof-construction function closures can occupy hundreds of megabytes. Since I am running a 32-bit Coq limited to 2 gigabytes and with a copying garbage collector, this consumes most of memory.

One solution, readily available, is to run 64-bit Coq on my 32-gigabyte desktop computer. But I would like to think that verifying a program as small as SHA-256 could readily be done on an ordinary laptop. Two other solutions to this problem are potentially available:

(1) Program the symbolic execution using computational reflection. That is, write a functional program in Coq to do symbolic execution of the Verifiable C program logic, prove it correct in Coq, and then apply it to C programs such as SHA-256. During the execution of such a program, Coq does not build proof traces. The VST project is already working on this approach [Appel et al. 2014, Chapters 25,46,47].
(2) Coq does not necessarily need to build proof terms as data structures. The Edinburgh LCF proof assistant in 1979 demonstrated a technique for using a small trustworthy proof-checking kernel using an abstract-data-type interface rather than a proof-term data structures [Gordon et al. 1979]. Some modern systems such as HOL and Isabelle/HOL also use this ADT approach. Using ADTs instead of proof terms would solve the memory problem, but it would require substantial change inside Coq.

10. IS IT REALLY OPENSSL?

Verifiable C is a subset of the C language; certain program transformations are needed before applying the program logic. Therefore I modified the OpenSSL implementation of SHA-256 in these ways:

(1) OpenSSL is heavily macro-ized so that the same source file generates SHA-224, SHA-256, and other instantiations. I expanded the macros and included header files just enough to specialize to SHA-256.
(2) Verifiable C prohibits side effects inside subexpressions; I broke these into separate statements.
(3) Verifiable C prohibits memory references inside subexpressions, and requires that each assignment statement have at most one memory reference at top level. This requires *local* rewriting, often with the introduction of a temporary variable.
(4) The current prototype Verifiable C requires a return statement instead of a fall-through at the end of a function, so I added some return statements.

The first two of these are handled automatically by the compiler *before* applying the program logic, so they do not require any manual changes to the program. The third one could also be handled by the compiler but is not at present. The last one will be remedied in the near future.

Still, by instantiating some macros I made my proof task easier, at the cost of limiting the generality of my result to the 256-bit case.

11. COMPOSING THIS PROOF WITH OTHERS

Verifications of individual components can suffer from the problem that the size of the specification can be larger than the size of the program. The machine-checked proof removes the program from the trusted base, but if the specification is as big as the program, what have we gained? The answer can come in the composition of systems. When we compose the SHA-256 proof with the Verifiable C proof with the CompCert proof, the entire specification of C drops out of the trusted base (as explained in §8).

At the other end, we should connect SHA-256 with its application, for example in the HMAC protocol for cryptographic authentication. HMAC calls upon a cryptographic hash function (such as SHA-256).

Desired claim: A particular implementation A of HMAC in the C language is a key-selected pseudorandom function (PRF) on the message.

Proof structure:

(1) The C program A, which calls upon SHA-256, correctly implements the functional specification of HMAC. (Future work: To be formalized and proved using techniques similar to those described in this paper.)
(2) The functional spec of HMAC (indexed by a randomly chosen key) gives a PRF, provided that the underlying hash primitive is a Merkle-Damgård hash construction applied to a PRF compression function. Proof: Future work based (for example) on Gaži et al. [2014] or Bellare *et al.* [1996; 2006] but fully formalized in Coq.[10]
(3) **OpenSSL's SHA-256 correctly implements the functional spec of SHA-256.** (This paper.)
(4) The functional spec of SHA-256 is a Merkle-Damgård hash construction. (Provable from the functional spec described in this paper; future work.)
(5) The compression function underlying SHA-256 is a PRF.[11] Oops! Nobody knows how to prove that SHA-256's compression function is a PRF. For now, the world survives on the fact that nobody knows how (without knowing the key) to distinguish it from a random function.

So this is a chain of proofs with a hole. Fortunately, the hole is in *only* the place where symmetric crypto always has a hole: the crypto properties of the symmetric-key primitive. This hole is bounded very closely by the *functional specifications* of both SHA-256 (169 lines of Coq) and of one-way functions. All the rest—the messy parts in C—are not in the trusted base, their connection is *proved* with machine-checked proofs.

That's *almost* true—but there's really one more thing. Eventually the end user has to call the HMAC function, from a C program. That user will need a specification related to C-language calling conventions. In this paper I have shown what such a specification looks like for SHA-256: the definition SHA256_spec in §8. This is not too large or complex; the specification of HMAC would be similar.

[10]There are rumors that something like this has been done in CertiCrypt, but no publication describes a CertiCrypt proof of HMAC or NMAC. There is a brief description of an EasyCrypt proof of NMAC in [Barthe et al. 2012], but (unlike CertiCrypt) EasyCrypt is not foundational: "EasyCrypt was conceived as a front-end to the CertiCrypt framework. . . . Certification remains an important objective, although the proof-producing mechanism may fall temporarily out of sync with the development of EasyCrypt." [Barthe et al. 2012] It appears that EasyCrypt/CertiCrypt have been continuously out of sync since 2012.

[11]More precisely: Bellare's [2006] HMAC proof requires that the underlying hash function H is a Merkle-Damgård construction on a round function $R(x, m)$ such that: (A) the "dual family" of R is secure against related-key attacks, and (B) R is a pseudorandom function (PRF). That is, given a random unknown key x, it is computationally intractable to distinguish $\lambda m.R(x, m)$ from a randomly chosen function over m.

12. THE TRUSTED BASE

The assurance of the correctness of SHA-256 relies on a *trusted base,* a series of specifications and implementations that must be correct for the proof to be meaningful. That is, we must trust:

Calculus of Inductive Constructions. The logic underlying Coq must be consistent in order to trust proofs in it. Several refereed papers have been published giving consistency arguments for versions of this logic, but these papers have not fully tracked the particular logic implemented in Coq. In general, CiC is a stronger and more complex logic than some of its competitors such as HOL and LF, so there is more to trust here.

Axioms. The soundess proof of Verifiable C uses the axioms of (dependent) functional extensionality and propositional extensionality. Both of these axioms are consistent with Coq's core theory, that is, when they are added to CiC it is still impossible to prove *false.* But in 2013 it was discovered that propositional extensionality is unsound in Coq 8.4.[12] This is not an inherent problem with the consistency of the axiom, it is a bug in Coq's termination checking.[13] It is expected that near-future releases of Coq will be consistent with functional and propositional extensionality.

Coq kernel. Coq has a *kernel* that implements proof-checking (type-checking) for CiC. We must trust that this kernel is free of bugs. The kernel is between 10,000 and 11,000 lines of ML code.

OCaml compiler. The Coq kernel's ML compiler is compiled by a the OCaml compiler, which is tens of thousands of lines of code. That compiler is compiled by itself, leading to an infinite regression that cannot be fully trusted [Thompson 1984].

OCaml runtime. Coq, running as an OCaml-compiled binary, is serviced by the OCaml runtime system and garbage collector, written in C.

Functional specification of SHA-256. We must trust that I have correctly transcribed the FIPS 180-4 standard into Coq. However, if (in the future) we complete crypto proofs *about* the functional spec, then this element drops out of the trusted base, to some extent.

API specification of SHA-256. The intended relation of the functional spec to data structures at API function calls, what I have called the "API spec", must be right, otherwise I have proved the wrong thing.

Specification of CompCert. I have explained earlier that we do not need to trust CompCert's specification of the C language, since that "drops out" of the trusted base when composed with the soundness proof of the Verifiable C program logic. But we do need to trust that CompCert's specification of Intel x86 (IA-32) assembly language is correct.

Assembler. At present, there is no proved-correct assembler for CompCert. The transition from assembly language to machine language is done by the GNU assembler and linker. Proving correctness of an assembler is quite achievable [Wu et al. 2003].

Intel Core i7. The OCaml binary and garbage collector run in machine language. My computer is an Intel Core i7, and we must trust that Intel has correctly implemented the instruction-set architecture, for two reasons: First, we run Coq on it, so that affects confidence in the proof-checking. Second, we run the SHA-256 on it,

[12] Daniel Schepler, Maxime Dénès, Arthur Charguéraud, "Propositional extensionality is inconsistent in Coq," coq-club mailing list, 12 December 2013.

[13] "Just to reassure everyone having developments relying on these axioms, the problem does not seem too deep. There is a slight inconsistency in the way the guard checker handles unreachability hypotheses, but that should be easily fixed without too much impact on existing contributions (hopefully)." Maxime Dénès, coq-club mailing list, 12 December 2013.

and therefore the specification of the assembly language is part of the assumptions of the CompCert correctness proof.

This is a long chain of trust. One can do *much* better. The Foundational Proof-Carrying code project had a trusted base of less than 3000 lines of code, including axioms, proof-checking kernel, compiler, runtime, functional specification, API specification, and ISA specification [Wu et al. 2003] (and it avoided the Thompson paradox by not needing a compiler in the trusted base). But that was for a much less ambitious project (safety instead of correctness) in a much weaker logic (LF instead of Coq). Scaling those tiny-trusted-base techniques to VST would not be easy.

13. RELATED WORK IN CRYPTO

One can compare to previous work on several dimensions:

Specification. Is there a specification of the program's function? Is the specification be written in a pure functional (or relational) language, amenable to analysis in a proof assistant? (That is, *aside* from verification that an implementation satisfies the functional spec, can one reason about the functional spec *per se*?)
Implementation. Is the proof about an efficient implementation (e.g., in compiled C or Java), or only about a functional spec?
Foundational. Is there an end-to-end machine-checked proof from the foundations of logic, that the generated assembly code correctly implements the specification, without trusting the compiler (or equivalently, without trusting the programming-language specification)? (Where there is no implementation, this question does not even apply.)
Automatic. Does the verifier check or synthesize the crypto algorithm without much (or any) interactive (or scripted) human input or annotations?
General. Can the verifier handle all parts of a crypto algorithm (such as the management code in SHA256_Update), or only the parts where the number of input bits is fixed and the loops can be completely unrolled?

Specification, implementation, foundational, not ~~automatic~~, general:. The work described in this paper.

Specification, implementation, not ~~foundational~~, automatic, not ~~general~~:. Smith and Dill [2008] verified several block-cipher implementations written in Java, w.r.t. a functional spec written either in Java or in ACL2. They compiled to byte-code, then used a subset model of the JVM to generate straight-line code. This renders them immune to bugs in javac, but the JIT compiler (from byte-code to native code) is unverified (or, equivalently, their JVM spec is unverified). They prove the straight-line code equivalent to the straight-line code of the functional spec. Their verification is fully automatic, using rewrite rules to simplify and normalize arithmetic expressions—with rules for many special patterns that occur in crypto code, such as bitfield concatentation by shift-and-or. After rewriting, they use a SAT solver to compare the normalized expressions. Smith and Dill's method applies only where the number of input bits is fixed and the loops can be completely unrolled. Their verifier would likely be applicable to the SHA-256 block shuffle (sha256_block_data_order) function, but certainly not to the management code (SHA256_Update).

Limited specifi~~cation~~, implementation, not ~~foundational~~, automatic, not ~~general~~:. Cryptol [Erkok et al. 2009] generates C or VHDL directly from a functional specification, where the number of input bits is fixed and the loops can be completely unrolled. In fact, Cryptol *does* unroll the loop in sha256_block_data_order, leading to a program that is 1.5x faster than OpenSSL's standard implementation (when both are compiled

by gcc and run on Intel Core i7)[14]. The spec is in a Haskell dialect, not directly embeddable in any existing proof assistant; the synthesizer is not verified.

Specification, implementation, not ~~foundational~~, not ~~automatic~~, not ~~general~~:. Toma and Borrione [2005] used ACL2 to prove correctness of a VHDL implementation of the SHA-1 block-shuffle algorithm.

No ~~specification~~, implementation, not ~~foundational~~, automatic, general:. One can apply static analysis algorithms to C programs to learn whether they have memory-safety bugs such as buffer overruns. Many such analyses are both unsound (will miss some bugs) and incomplete (will report false-positives about bug-free programs). Even so, they can be very useful; but they do not attempt to prove functional correctness with respect to a specification.

Complementary work. In this paper I have concentrated on the verification that an implementation satisfies its functional specification. Complementary work establishes properties of the functional specs. For example, Duan et al. [2005] proved a property of the functional specs of several encryption algorithms: that decryption is the inverse of encryption. More relevant to SHA-256, Backes et al. [2012] verify mechanically (in EasyCrypt) that Merkle-Damgård constructions have certain security properties. Bellare [1996; 2006] gave the first proofs of NMAC/HMAC security (without a machine-checked proof); Gaži et al. [2014] prove PRF-security of NMAC/HMAC (without a machine-checked proof), based on fewer assumptions.

EasyCrypt. Almeida et al. [2013] describe the use of their EasyCrypt tool to verify the security of an implementation of the RSA-OAEP encryption scheme. A functional specification of RSA-OAEP is written in EasyCrypt, which then verifies its security properties. An unverified Python script translates the EasyCrypt specification to (an extension of) C; then an extension of CompCert compiles it to assembly language. Finally, a leakage tool verifies that the assembly-language program has no more program-counter leakage than the source code, i.e. that the compiled program's trace of conditional branches is no more informative to the adversary than the source code's.

The EasyCrypt verifier is not fully foundational; it is an OCaml program whose correctness is not proved. The translation from EasyCrypt to C is not foundational. The translation from C to assembly language is foundational, using CompCert. Programs must operate on fixed-size data blocks.

The leakage model is the Program Counter Model (trace of conditional branches), and there is a foundational checker (i.e., proved correct in Coq) that compiled programs leak no more PC-trace information than the source program. But other forms of leakage are not modeled. In particular, SHA-256 has a line of code (marked `/* keep it zeroed */`) whose entire purpose is to reduce leakage of message fragments through deallocated memory; but that kind of leakage channel is not modeled in EasyCrypt.

EasyCrypt's C code relies on bignum library functions called through a nonstandard specification interface—nonstandard, because standard CompCert (through the current version, 2.3) has no way to handle external function calls that receive or return results in memory. But EasyCrypt provides no mechanism by which these functions can be proved correct, nor does it give a mechanized proof theory for this custom specification interface.

In summary, Almeida *et al.* attack two problems that are exactly complementary to the work I have done. EasyCrypt allows reasoning about crypto properties of the

[14]Aaron Tomb, Galois.com, personal communication, 13 January 2014.

functional spec; and they extend CompCert's compilation-correctness guarantees with a guarantee about a particular side channel. They do not reason about the semantic relation between the functional spec and the C program (either the main algorithm or the bignum library).

14. RELATED WORK IN C VERIFICATION

There are many program analysis tools for C. Most of them do not address functional specification or functional correctness, and most are unsound and incomplete. Nonetheless, they are very useful in practice: C static-analysis tools are a billion-dollar-a-year industry.[15]

Foundational formal verification of C programs has only recently been possible. The most significant such works are both operating-system kernels: seL4 [Klein et al. 2009] and CertiKOS [Gu et al. 2011]. Both proofs are refinement proofs between functional specifications and operational semantics. Both proofs are done in higher-order logics: seL4 in Isabelle/HOL and CertiKOS in Coq. Each of these projects verifies a significantly larger C program than the SHA-256 program I describe here.

Neither of their proof frameworks use separation logic, neither can accommodate the use of addressable local variables in C, and neither can handle function-pointers or higher-order specifications. This means that the OpenSSL SHA-256 program could not be proved in these frameworks, because it uses addressable local variables. A minor adjustment of the C program—moving the X array into the SHA256state_st structure—would eliminate the use of addressable locals, however.

SHA-256 does not use function pointers. However, OpenSSL uses function pointers in its "engines" mechanism, an object-oriented style of programming that dynamically connects components together—for example, HMAC and SHA. The *Verifiable C* program logic, with higher-order separation logic, is capable of reasoning about such object-oriented patterns in C [Appel et al. 2014, Chapter 29].

The C semantics used in the original seL4 proof was not connected to a C compiler (e.g., it is not the CompCert C semantics), so the entire C semantics is part of the trusted base of the seL4 proof. More recent work removes C from the trusted base: Sewell et al. [2013] perform translation validation for `gcc 4.5.1` by decompiling ARM code to logical graphs, then using a combination of carefully tuned heuristic proof search and SMT solving to find a proof of equivalence between source program and machine language.

CertiKOS is proved correct with respect to the CompCert C semantics, so (as in my SHA-256 proof) this C semantics drops out of the trusted base.

Both seL4 and CertiKOS are newly written C programs designed for verification. In principle, this is the right way to do things: it can be difficult to verify pre-existing programs. However, there are times when it's important to be able to do so. Aircraft manufacturers, who have code bases already certified (and trusted) for use in passenger jets, should not be asked to rewrite their fly-by-wire software just so that they can apply new and better verification techniques. And, to the extent that the security community has come to trust nonfunctional properties of OpenSSL—lack of timing channels, fault injection resistance, compatibility with many C compilers—this trust cannot necessarily be transferred to new implementations.

15. CONCLUSION

Functional correctness verification of C programs has important applications in computer security. Correctness has the corollary of memory safety, which is valuable in

[15] Andy Chou, "From the Trenches: Static Analysis in Industry", invited talk at POPL 2014, January 24, 2014.

itself. But in the implementatation of protection mechanisms (such as operating systems, encryption, authentication), safety is not enough: correctness is what guarantees that these mechanisms actually secure the systems that they are supposed to protect.

C is not friendly to program verification: it has tricky corners, one needs to keep track of many side conditions. Nonetheless it is possible to do full formal verification of C programs. Previous results have demonstrated this for operating-system microkernels [Klein et al. 2009] (though not with the verified connection to a verified compiler). In such results, the C program is typically constructed anew with a design particularly suited for the verification task.

In this project I demonstrated that one can verify a program *as it is*. This is valuable because widely used open-source cryptographic primitives have many relevant properties other than functional correctness. For example, the comment `/* keep it zeroed */` in the Update function is attached to a line that has no functional impact, but might reduce information flow through deallocated variables. My program logic cannot prove that this line reduces side channels, but at least I can prove that it does not impair functional correctness. Whatever assurance and confidence the community has gained in this program will only be increased by this verification.

At the same time, these cryptographic primitives have many implementation variants (using machine-dependent instructions). Small variations of this proof can serve to prove all of them equivalent to the same functional specification, even if there are not "many eyes" on every single one of them to keep the bugs shallow.

REFERENCES

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2013. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications security*. ACM, 1217–1230.

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge.

Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *First International Conf. on Certified Programs and Proofs*.

Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Grégoire, César Kunz, Malte Skoruppa, and Santiago Zanella Béguelin. 2012. Verified security of Merkle-Damgård. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE, 354–368.

Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. 2012. Computer-aided cryptographic proofs. In *Interactive Theorem Proving*. Springer, 11–27.

Mihir Bellare. 2006. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology-CRYPTO 2006*. Springer, 602–619.

Mihir Bellare, Ran Canetti, and Hugo Krawczyk. 1996. Keying hash functions for message authentication. In *Advances in CryptologyCRYPTO96*. Springer, 1–15.

Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. 2011. Modular SMT Proofs for Fast Reflexive Checking inside Coq. In *First International Conf. on Certified Programs and Proofs*.

Lindsey Bever. 2014. Major bug called 'Heartbleed' exposes Internet data. *Washington Post* (9 April 2014).

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-End Verification of Stack-Space Bounds for C Programs. In *In Proc. 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*.

Jianjun Duan, Joe Hurd, Guodong Li, Scott Owens, Konrad Slind, and Junxing Zhang. 2005. Functional correctness proofs of encryption algorithms. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 519–533.

L. Erkok, Magnus Carlsson, and Adam Wick. 2009. Hardware/software co-verification of cryptographic algorithms using Cryptol. In *Formal Methods in Computer-Aided Design, 2009 (FMCAD'09)*. IEEE, 188–191.

FIPS 2012. *Secure Hash Standard (SHS)*. Technical Report FIPS PUB 180-4. Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD.

Peter Gaži, Krzysztof Pietrzak, and Michal Rybár. 2014. The Exact PRF-Security of NMAC and HMAC. In *Advances in Cryptology–CRYPTO 2014*. Springer, 113–130.

Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF: A Mechanised Logic of Computation*. Lecture Notes in Computer Science, Vol. 78. Springer-Verlag, New York.

David Greenaway, June Andronick, and Gerwin Klein. 2012. Bridging the gap: Automatic verified abstraction of C. In *Interactive Theorem Proving*. Springer, 99–115.

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys'11)*. ACM, Article 3, 5 pages. 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, and others. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

John Regehr. 2014. A New Development for Coverity and Heartbleed. *Embedded in Academia* (12 April 2014). blog.regehr.org/archives/1128

Bruce Schneier. 1999. Open Source and Security. *Crypto-Gram Newsletter* (15 Sept. 1999).

Bruce Schneier. 2013. How to Remain Secure Against the NSA. *Crypto-Gram Newsletter* (15 Sept. 2013).

Thomas A. L. Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. *ACM SIGPLAN Notices* 48, 6 (2013), 471–482.

Eric W. Smith and David L. Dill. 2008. Automatic formal verification of block cipher implementations. In *Formal Methods in Computer-Aided Design (FMCAD'08)*. IEEE, 1–7.

Ken Thompson. 1984. Reflections on Trusting Trust. 27, 8 (1984), 761–763.

Diana Toma and Dominique Borrione. 2005. Formal verification of a SHA-1 circuit core using ACL2. In *Theorem Proving in Higher Order Logics*. Springer, 326–341.

Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *Proceedings 24th ACM Symposium on Operating Systems Principles*. ACM, 260–275.

Dinghao Wu, Andrew W. Appel, and Aaron Stump. 2003. Foundational Proof Checkers with Small Witnesses. In *PPDP*. 264–274.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2012. Finding and understanding bugs in C compilers. *ACM SIGPLAN Notices* 47, 6 (2012), 283–294.

APPENDIX

A. SHA-256, C PROGRAM ADAPTED FROM OPENSSL

```
/* Adapted 2013 from OpenSSL098 crypto/sha/sha256.c     Copyright (c) 2004 The OpenSSL
 * Project.  All rights reserved according to the OpenSSL license.
 */

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

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

#define HOST_c2l(c,l)   \
   (l=(unsigned long)(__builtin_read32_reversed (((unsigned int *)c))),c+=4,l)

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

#define SHA_LONG unsigned int

#define SHA_LBLOCK      16
#define SHA_CBLOCK      (SHA_LBLOCK*4)
   /* SHA treats input data as a contiguous array of 32 bit wide big-endian values. */
#define SHA_LAST_BLOCK  (SHA_CBLOCK-8)
#define SHA_DIGEST_LENGTH 20

#define SHA256_DIGEST_LENGTH    32

typedef struct SHA256state_st {
        SHA_LONG h[8];
        SHA_LONG Nl,Nh;
        unsigned char data[SHA_CBLOCK];
        unsigned int num;
        } SHA256_CTX;

#define MD32_REG_T long
#define ROTATE(a,n) (((a)<<(n))|(((a)&0xffffffff)>>(32-(n))))

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

/* FIPS specification refers to right rotations, while our ROTATE macro is left one.
 * This is why you might notice that rotation coefficients differ from those
 * observed in FIPS document by 32-N...
 */
#define Sigma0(x)   (ROTATE((x),30) ^ ROTATE((x),19) ^ ROTATE((x),10))
#define Sigma1(x)   (ROTATE((x),26) ^ ROTATE((x),21) ^ ROTATE((x),7))
#define sigma0(x)   (ROTATE((x),25) ^ ROTATE((x),14) ^ ((x)>>3))
#define sigma1(x)   (ROTATE((x),15) ^ ROTATE((x),13) ^ ((x)>>10))

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

void sha256_block_data_order (SHA256_CTX *ctx, const void *in) {
  unsigned MD32_REG_T a,b,c,d,e,f,g,h, s0,s1,T1,T2,t;
  SHA_LONG      X[16],l,Ki;
```

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

  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];

  for (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;    e = d + T1;
      d = c;    c = b;    b = a;    a = T1 + T2;
  }

  for (;i<64;i++) {
      s0 = X[(i+1)&0x0f];    s0 = sigma0(s0);
      s1 = X[(i+14)&0x0f];   s1 = sigma1(s1);
      T1 = X[i&0xf];
      t = X[(i+9)&0xf];
      T1 += s0 + s1 + t;
      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;    e = d + T1;
      d = c;    c = b;    b = a;    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;
  return;
}

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->Nh=0;
  c->num=0;
  return;
}

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

```
  c->Nl=l; c->Nh=cNh;
  return;
}

void SHA256_Update (SHA256_CTX *c,
                const void *data_, size_t len) {
  const unsigned char *data=data_;
  unsigned char *p;
  size_t   n, fragment;

  SHA256_addlength(c, len);
  n = c->num;
  p=c->data;
  if (n != 0)    {
    fragment = SHA_CBLOCK-n;
    if (len >= fragment)  {
       memcpy (p+n,data,fragment);
       sha256_block_data_order (c,p);
       data  += fragment;
       len   -= fragment;
       memset (p,0,SHA_CBLOCK); /* keep it zeroed */
    }
    else  {
       memcpy (p+n,data,len);
       c->num = n+(unsigned int)len;
       return;
    }
  }
  while (len >= SHA_CBLOCK) {
      sha256_block_data_order (c,data);
      data += SHA_CBLOCK;
      len  -= SHA_CBLOCK;
  }
  c->num=len;
  if (len != 0) {
      memcpy (p,data,len);
  }
  return;
}

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

  p[n] = 0x80; /* there is always room for one */
  n++;

  if (n > (SHA_CBLOCK-8)) {
        memset (p+n,0,SHA_CBLOCK-n);
        n=0;
        sha256_block_data_order (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;
  sha256_block_data_order (c,p);
  c->num=0;
  memset (p,0,SHA_CBLOCK);
  {unsigned long ll;
   unsigned int  xn;
   for (xn=0;xn<SHA256_DIGEST_LENGTH/4;xn++)
        {   ll=(c)->h[xn]; HOST_l2c(ll,md);   }
  }
  return;
}

void SHA256(const unsigned char *d,
            size_t n, unsigned char *md) {
  SHA256_CTX c;
  SHA256_Init(&c);
  SHA256_Update(&c,d,n);
  SHA256_Final(md,&c);
  return;
}
```

B. THE SPECIFICATION

Definition big_endian_integer (contents: Z → int) : int :=
  Int.or (Int.shl (contents 0) (Int.repr 24))
  (Int.or (Int.shl (contents 1) (Int.repr 16))
   (Int.or (Int.shl (contents 2) (Int.repr 8))
      (contents 3))).

Definition LBLOCKz : Z := 16. *(∗ length of a block, in 32-bit ints ∗)*
Definition CBLOCKz : Z := 64. *(∗ length of a block, in characters ∗)*

Definition s256state := (list val ∗ (val ∗ (val ∗ (list val ∗ val))))%type.
Definition s256_h (s: s256state) := fst s.
Definition s256_Nl (s: s256state) := fst (snd s).
Definition s256_Nh (s: s256state) := fst (snd (snd s)).
Definition s256_data (s: s256state) := fst (snd (snd (snd s))).
Definition s256_num (s: s256state) := snd (snd (snd (snd s))).

Inductive s256abs := *(∗ SHA-256 abstract state ∗)*
 S256abs: ∀ (hashed: list int) *(∗ words hashed, so far ∗)*
            (data: list Z), *(∗ bytes in the partial block not yet hashed ∗)*
            s256abs.

Definition s256a_regs (a: s256abs) : list int :=
 match a **with** S256abs hashed data ⇒ hash_blocks init_registers hashed **end**.

Definition s256a_len (a: s256abs) : Z :=
  match a **with** S256abs hashed data ⇒ (Zlength hashed ∗ 4 + Zlength data) ∗ 8
end%Z.

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

Definition s256_relate (a: s256abs) (r: s256state) : Prop :=
      match a with S256abs hashed data ⇒
          s256_h r = map Vint (hash_blocks init_registers hashed)
        ∧ (∃ hi, ∃ lo, s256_Nh r = Vint hi ∧ s256_Nl r = Vint lo ∧
              (Zlength hashed * 4 + Zlength data)*8 = hilo hi lo)%Z
        ∧ s256_data r = map Vint (map Int.repr data)
        ∧ (Zlength data < CBLOCKz ∧ Forall isbyteZ data)
        ∧ (LBLOCKz | Zlength hashed)
        ∧ s256_num r = Vint (Int.repr (Zlength data))
      end.

Definition init_s256abs : s256abs := S256abs nil nil.

Definition sha_finish (a: s256abs) : list Z :=
 match a with S256abs hashed data ⇒ SHA_256 (intlist_to_Zlist hashed ++ data) end.

Definition cVint (f: Z → int) (i: Z) := Vint (f i).

Definition sha256_length (len: Z) (c: val) : mpred :=
   EX lo:int, EX hi:int,
      !! (hilo hi lo = len) &&
      (field_at Tsh t_struct_SHA256state_st _Nl (Vint lo) c *
       field_at Tsh t_struct_SHA256state_st _Nh (Vint hi) c).

Definition sha256state_ (a: s256abs) (c: val) : mpred :=
   EX r:s256state, !! s256_relate a r && data_at Tsh t_struct_SHA256state_st r c.

Definition tuints (vl: list int) := ZnthV tuint (map Vint vl).
Definition tuchars (vl: list int) := ZnthV tuchar (map Vint vl).

Definition data_block (sh: share) (contents: list Z) :=
  !! Forall isbyteZ contents &&
  array_at tuchar sh (tuchars (map Int.repr contents))
             0 (Zlength contents).

Definition __builtin_read32_reversed_spec :=
 DECLARE ___builtin_read32_reversed
  WITH p: val, sh: share, contents: Z → int
  PRE [ 1 OF tptr tuint ]
      PROP() LOCAL (`(eq p) (eval_id 1))
      SEP (`(array_at tuchar sh (cVint contents) 0 4 p))
  POST [ tuint ]
      local (`(eq (Vint (big_endian_integer contents))) retval) &&
      `(array_at tuchar sh (cVint contents) 0 4 p).

Definition __builtin_write32_reversed_spec :=
 DECLARE ___builtin_write32_reversed
  WITH p: val, sh: share, contents: Z → int
```

```
  PRE [ 1 OF tptr tuint, 2 OF tuint ]
     PROP(writable_share sh)
     LOCAL (`(eq p) (eval_id 1);
              `(eq (Vint(big_endian_integer contents))) (eval_id 2))
     SEP (`(memory_block sh (Int.repr 4) p))
  POST [ tvoid ]
     `(array_at tuchar sh (cVint contents) 0 4 p).

Definition memcpy_spec := (* elided *)
Definition memset_spec := (* elided *)

Definition K_vector : environ → mpred :=
    array_at tuint Tsh (tuints K) 0 (Zlength K).

Definition sha256_block_data_order_spec :=
  DECLARE _sha256_block_data_order
  WITH hashed: list int, b: list int, ctx : val, data: val, sh: share
  PRE [ _ctx OF tptr t_struct_SHA256state_st, _in OF tptr tvoid ]
     PROP(Zlength b = LBLOCKz; (LBLOCKz | Zlength hashed))
     LOCAL (`(eq ctx) (eval_id _ctx); `(eq data) (eval_id _in))
     SEP (`(array_at tuint Tsh
               (tuints (hash_blocks init_registers hashed)) 0 8 ctx);
        `(data_block sh (intlist_to_Zlist b) data);
        `K_vector (eval_var _K256 (tarray tuint 64)))
  POST [ tvoid ]
     (`(array_at tuint Tsh
           (tuints (hash_blocks init_registers (hashed++b))) 0 8 ctx) *
      `(data_block sh (intlist_to_Zlist b) data) *
      `K_vector (eval_var _K256 (tarray tuint 64))).

Definition SHA256_addlength_spec :=
 DECLARE _SHA256_addlength
 WITH len : Z, c: val, n: Z
 PRE [ _c OF tptr t_struct_SHA256state_st , _len OF tuint ]
    PROP ( 0 <= n+len*8 < two_p 64)
    LOCAL (`(eq len) (`Int.unsigned (`force_int (eval_id _len)));
                `(eq c) (eval_id _c))
    SEP (`(sha256_length n c))
 POST [ tvoid ]
    `(sha256_length (n+len*8) c).

Definition SHA256_Init_spec :=
  DECLARE _SHA256_Init
   WITH c : val
   PRE [ _c OF tptr t_struct_SHA256state_st ]
          PROP () LOCAL (`(eq c) (eval_id _c))
          SEP(`(data_at_ Tsh t_struct_SHA256state_st c))
  POST [ tvoid ]
           (`(sha256state_ init_s256abs c)).

Inductive update_abs: list Z → s256abs → s256abs → Prop :=
 Update_abs:
```

```
  (∀ msg hashed blocks oldfrag newfrag,
       Zlength oldfrag < CBLOCKz →
       Zlength newfrag < CBLOCKz →
      (LBLOCKz | Zlength hashed) →
      (LBLOCKz | Zlength blocks) →
      oldfrag++msg = intlist_to_Zlist blocks ++ newfrag →
  update_abs msg (S256abs hashed oldfrag)
                 (S256abs (hashed++blocks) newfrag)).

Definition SHA256_Update_spec :=
  DECLARE _SHA256_Update
   WITH a: s256abs, data: list Z, c : val, d: val, sh: share, len : nat
   PRE [ _c OF tptr t_struct_SHA256state_st, _data_ OF tptr tvoid, _len OF tuint ]
         PROP ((len <= length data)%nat;
                  (s256a_len a + Z.of_nat len * 8 < two_p 64)%Z)
         LOCAL (`(eq c) (eval_id _c); `(eq d) (eval_id _data_);
                `(eq (Z.of_nat len))
                   (`Int.unsigned (`force_int (eval_id _len))))
         SEP(`K_vector (eval_var _K256 (tarray tuint 64));
             `(sha256state_ a c); `(data_block sh data d))
  POST [ tvoid ]
         EX a':_,
          PROP (update_abs (firstn len data) a a') LOCAL ()
          SEP(`K_vector (eval_var _K256 (tarray tuint 64));
              `(sha256state_ a' c); `(data_block sh data d)).

Definition SHA256_Final_spec :=
  DECLARE _SHA256_Final
   WITH a: s256abs, md: val, c : val, shmd: share, sh: share
   PRE [ _md OF tptr tuchar, _c OF tptr t_struct_SHA256state_st ]
         PROP (writable_share shmd)
         LOCAL (`(eq md) (eval_id _md); `(eq c) (eval_id _c))
         SEP(`K_vector (eval_var _K256 (tarray tuint 64));
             `(sha256state_ a c);
             `(memory_block shmd (Int.repr 32) md))
  POST [ tvoid ]
         PROP () LOCAL ()
         SEP(`K_vector (eval_var _K256 (tarray tuint 64));
             `(data_at_ Tsh t_struct_SHA256state_st c);
             `(data_block shmd (sha_finish a) md)).
```