verifying SHA using VST

Freek Wiedijk

last paper in the reading list of
Type Theory & Coq
2015–2016
Radboud University Nijmegen

June 16, 2016

SHA and VST

- SHA =
  Secure Hash Algorithm

- VST =
  Verified Software Toolchain

papers

papers by Andrew Appel:

- Verification of a Cryptographic Primitive: SHA-256
  TOPLAS = ACM Transactions on Programming Languages and Systems

  April 2015

- Second Edition: Verification of a Cryptographic Primitive: SHA-256
  updated from VST 1.0 to VST 1.6

papers

papers by Andrew Appel:

- Verification of a Cryptographic Primitive: SHA-256
  TOPLAS = ACM Transactions on Programming Languages and Systems

  April 2015

- Second Edition: Verification of a Cryptographic Primitive: SHA-256
  updated from VST 1.0 to VST 1.6

- Modular Verification for Computer Security
  CSF 2016 = Computer Security Foundations Symposium

  June 2016

recap reading list

overview

- **imp**
  - big-step operational semantics
  - small-step operational semantics
  - Hoare logic
  - verification condition generator
- **CompCert**
  - idem for C
- **VST**
  - separation logic
  - symbolic execution

imp

syntax:

$$a ::= n \mid x \mid (a_1 + a_2) \mid (a_1 - a_2) \mid (a_1 \cdot a_2)$$
$$b ::= a_1 = a_2 \mid a_1 < a_2 \mid \top \mid \neg b \mid (b_1 \wedge b_2)$$
$$c ::= \text{skip} \mid x := a \mid (c_1; c_2) \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi} \mid \text{while } b \text{ do } c \text{ od}$$

example:

$$\begin{array}{l}(i := 1;\\ \ f := 1);\\ \text{while } i < n \text{ do}\\ \quad i := i + 1;\\ \quad f := f \cdot i\\ \text{od}\end{array}$$

big-step operational semantics

= natural semantics
Gilles Kahn

relation:

$$(c, s) \Downarrow s'$$

some representative rules:

$$\frac{(a, s) \Downarrow n}{(x := a, s) \Downarrow s[x \mapsto n]}$$

$$\frac{(c_1, s) \Downarrow s' \quad (c_2, s') \Downarrow s''}{(c_1; c_2, s) \Downarrow s''}$$

$$\frac{(b, s) \Downarrow \top \quad (c, s) \Downarrow s' \quad (\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od}, s') \Downarrow s''}{(\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od}, s) \Downarrow s''}$$

$$\frac{(b, s) \Downarrow \bot}{(\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od}, s) \Downarrow s}$$

small-step operational semantics

= structural operational semantics = SOS
Gordon Plotkin

relations:

$$(c, s) \to (c', s') \qquad (c, s) \to^* (c', s')$$

some representative rules:

$$\frac{(a, s) \to (a', s)}{(x := a, s) \to (x := a', s)}$$

$$\frac{}{(x := n, s) \to (\mathsf{skip}, s[x \mapsto n])}$$

$$\frac{(c_1, s) \to (c_1', s')}{(c_1; c_2, s) \to (c_1'; c_2, s')}$$

$$\frac{}{(\mathsf{skip}; c_2, s) \to (c_2, s)}$$

$$\frac{}{(\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od}, s) \to (\mathsf{if}\ b\ \mathsf{then}\ c\,;\ \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}, s)}$$

Hoare logic

= axiomatic semantics
Tony Hoare

Hoare triple:

$$\{P\}\, c\, \{Q\}$$

some representative rules:

$$\frac{}{\{Q[x := a]\}\, x := a\, \{Q\}}$$

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

$$\frac{\{P \wedge b\}\, c\, \{P\}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od}\, \{P \wedge \neg b\}}$$

$$\frac{P \Rightarrow P' \quad \{P'\}\, c\, \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\, c\, \{Q\}}$$

verification conditions from weakest preconditions

predicate transformer semantics
Edsger Dijkstra

imp with annotations:

$$\overline{c} ::= \{P\} \mid \text{skip} \mid x := a \mid (\overline{c}_1; \overline{c}_2) \mid \text{if } b \text{ then } \overline{c}_1 \text{ else } \overline{c}_2 \text{ fi} \mid \text{while } b \text{ do } \{P\}\ \overline{c} \text{ od}$$

verification condition and weakest precondition:

$$vc(\{P\}\ \overline{c}\ \{Q\}) = (P \Rightarrow wp(\overline{c}, Q))$$

some representative cases:

$$\begin{aligned} wp(\{P\}, Q) &= P \wedge Q \\ wp(x := a, Q) &= Q[x := a] \\ wp(\overline{c}_1; \overline{c}_2, Q) &= wp(\overline{c}_1, wp(\overline{c}_2, Q)) \\ wp(\text{while } b \text{ do } \{P\}\ \overline{c} \text{ od}, Q) &= P \wedge (P \wedge b \Rightarrow wp(\overline{c}, P)) \wedge (P \wedge \neg b \Rightarrow Q) \end{aligned}$$

CompCert

Xavier Leroy, INRIA, France

CompCert = idem for C

- C to Clight translator in OCaml
- optimizing Clight compiler as a Coq function
- Coq code extracted to OCaml

- operational semantics of Clight in Coq
- operational semantics of assembly in Coq
- compiler proved correct in Coq

separation logic

Hoare logic for pointers in memory
John Reynolds and Peter O'Hearn

$$\text{state} = \text{store} \times \text{heap}$$
$$\text{store} = \text{ident} \to \mathbb{Z}$$
$$\text{heap} = \mathbb{Z} \rightharpoonup \mathbb{Z}$$

separation logic assertions:

$$\text{emp}$$
$$a_1 \mapsto a_2$$
$$P * Q$$

frame rule:

$$\frac{\{P\}\, c\, \{Q\}}{\{P * R\}\, c\, \{Q * R\}}$$

VST

Andrew Appel, Princeton, US

VST

= Verified Software Toolchain

= CompCert +

- separation logic
- semantics for separate compilation
- symbolic execution
  - Coq goal is a Hoare triple
  - tactics execute statements

SSL, TSL and OpenSSL

OpenSSL
= open source implementation of SSL and TLS protocols

used by majority of the web servers

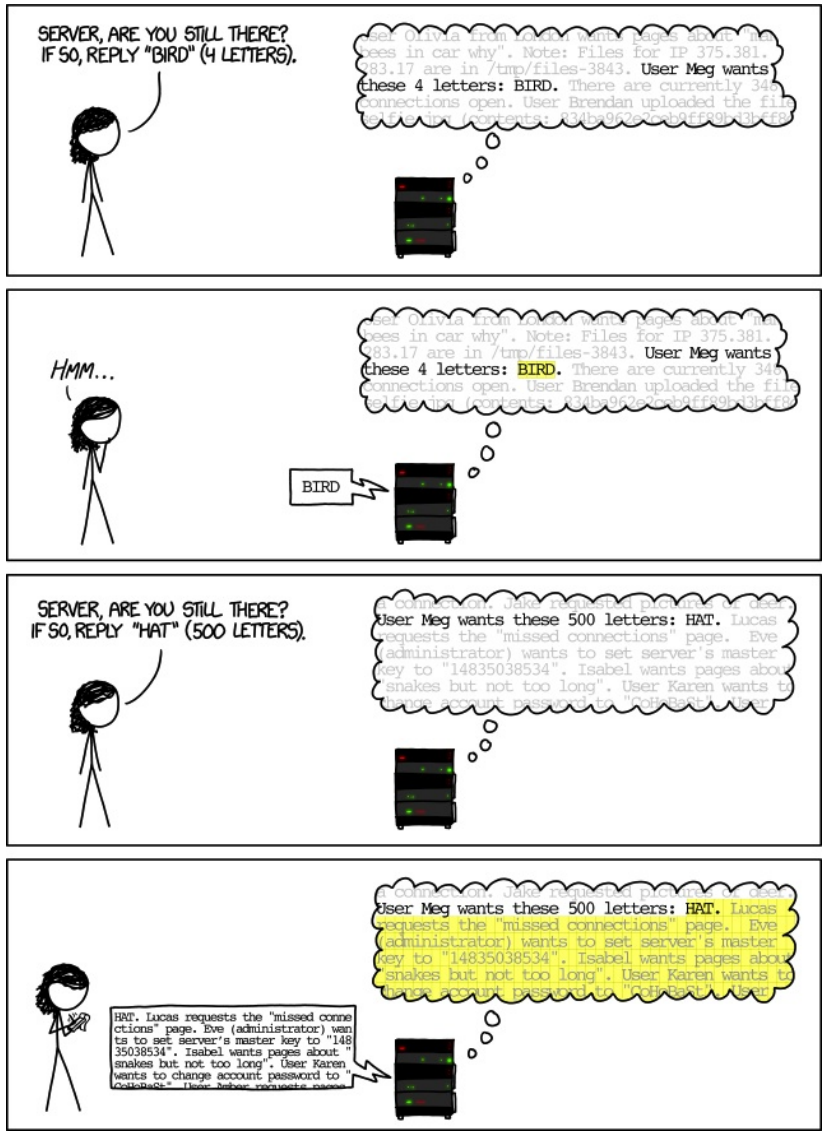
SSL = Secure Socket Layer
TLS = Transport Layer Security

secure communication on the internet

private connection: symmetric cryptography
identity checking: public-key cryptography
reliable connection

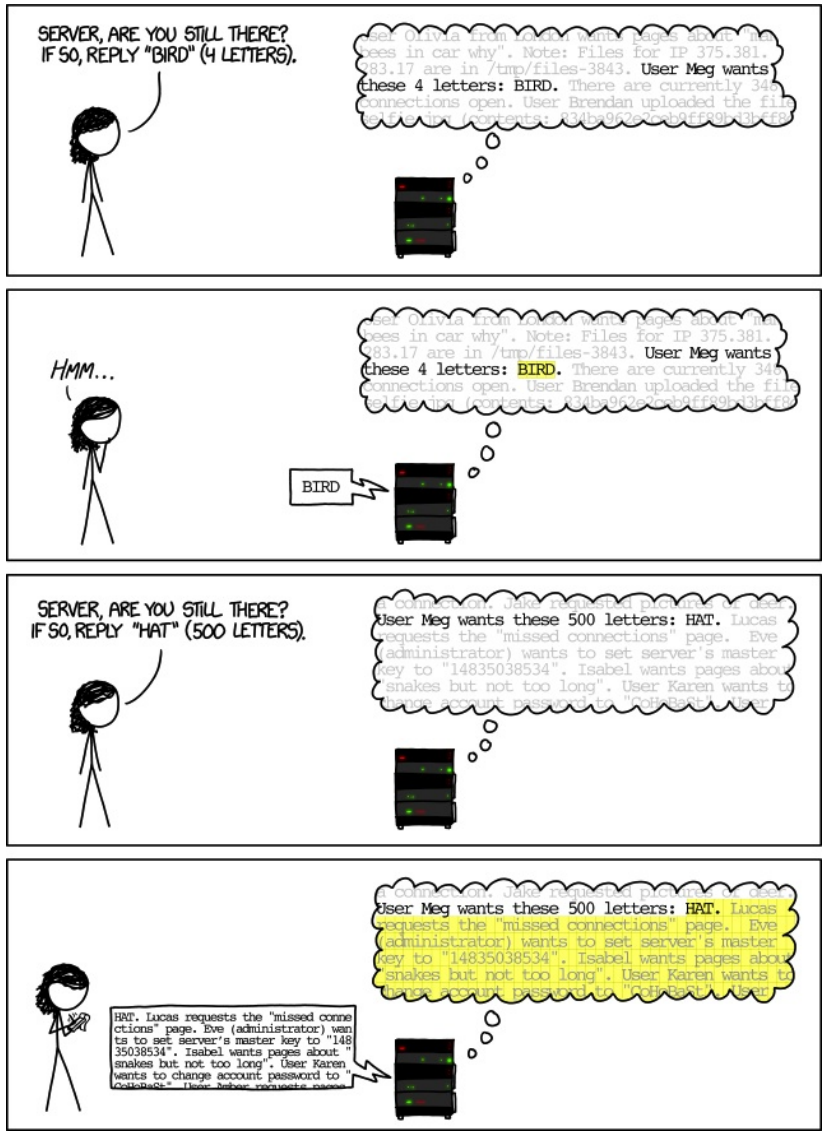
HTTPS = HTTP + TLS

heartbleed



April 2014

heartbleed



April 2014

fix is two lines in `ssl/d1_lib.c`:

```
if (HEARTBEAT_SIZE_STD
        (payload) > length)
    return 0;
/* silently discard per
        RFC 6520 sec. 4 */
```

cryptographic hashing

cryptographic hash function:

$$h : \{0,1\}^* \to \{0,1\}^{256}$$

four properties:

- $h(x)$ can be computed quickly
- given $h(x)$ finding a corresponding x is infeasible
- small change in x gives a large change in $h(x)$
- infeasible to find a collision: x_1 and x_2 with $h(x_1) = h(x_2)$

cryptographic hashing

cryptographic hash function:

$$h : \{0,1\}^* \to \{0,1\}^{256}$$

four properties:

- $h(x)$ can be computed quickly
- given $h(x)$ finding a corresponding x is infeasible
- small change in x gives a large change in $h(x)$
- infeasible to find a collision: x_1 and x_2 with $h(x_1) = h(x_2)$

examples:

$h(\texttt{"Lynx c.q. vos prikt bh: dag zwemjuf"}) =$
17c2f3484ab21559fa8d7bf3da97e3443b48a3466f3b8fa8210dbcefe99807a1

$h(\texttt{"Lynx c.q. vos prikt bh: dag zwemjuf!"}) =$
3530df7cc04da1f245eb92e5780610c5e0aa066a94ba17a66e2e310a64f1bd4d

SHA-256 and HMAC

SHA = Secure Hash Algorithm

SHA-0: 1993, SHA-1: 1995, SHA-2: 2001, SHA-3: 2015

SHA-0: collision known
SHA-1: collision unknown, but within range of supercomputers

SHA-2 = FIPS PUB 180-2 standard of NIST =
SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256

SHA-256: used by bitcoin

SHA-256 and HMAC

SHA = Secure Hash Algorithm

SHA-0: 1993, SHA-1: 1995, SHA-2: 2001, SHA-3: 2015

SHA-0: collision known
SHA-1: collision unknown, but within range of supercomputers

SHA-2 = FIPS PUB 180-2 standard of NIST =
SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256

SHA-256: used by bitcoin

HMAC = Hash-based Message Authentication Code

- authenticity: message came from sender
- integrity: message has not been tampered with

VST example: verifying factorial

workflow

VST example: verifying factorial

workflow

- `fac.c`
  C program being verified
  - `fac`
    C function calculating factorial

VST example: verifying factorial

workflow

- `fac.c`
  C program being verified
  - `fac`
    C function calculating factorial
- `fac.v`
  Clight version as a generated Coq file

VST example: verifying factorial

workflow

- `fac.c`
  C program being verified
  - `fac`
    C function calculating factorial
- `fac.v`
  Clight version as a generated Coq file
- `verif_fac.v`
  Coq file with the verification

VST example: verifying factorial

workflow

- `fac.c`
  C program being verified
  - `fac`
    C function calculating factorial
- `fac.v`
  Clight version as a generated Coq file
- `verif_fac.v`
  Coq file with the verification
  - `FAC`
    Coq functional program for each function in `fac.c`
  - `fac_spec`
    specification relating each function in `fac.c` to its Coq version
  - `body_fac`
    verification of correctness of each function in `fac.c`

fac.c

10 lines of C
calculates the factorial function

```
int
fac(int n)
{
  int i, f;

  f = i = 1;
  while (i < n)
    f *= ++i;
  return f;
}
```

fac.v

320 lines of Coq, **generated** from `fac.c` by CompCert's `clightgen`

```
...
Definition _n : ident := 45%positive. ...
Definition _fac : ident := 48%positive. ...

Definition f_fac := {|
  fn_return := tint;
  fn_callconv := cc_default;
  fn_params := ((_n, tint) :: nil);
  fn_vars := nil;
  fn_temps := ((_i, tint) :: (_f, tint) :: (51%positive, tint) ::
               (50%positive, tint) :: nil);
  fn_body := (Ssequence (Ssequence ... ...) ...)
|}.
...
Definition prog : Clight.program := {|
prog_defs := (... :: (_fac, Gfun(Internal, f_fac)) :: nil);
...
|}.
```

verif_fac.v

59 lines of Coq
checking time: 75 seconds

verif_fac.v

59 lines of Coq
checking time: 75 seconds

full code in these slides

starts with imports:

```
Require Import floyd.proofauto.
Require Import Coqlib.
Require Import Recdef.
```

FAC

implementation of factorial in Coq using `Function`
(recursion on `Acc` well-foundedness predicate):

```
Function FAC (i : Z) {measure Z.to_nat i} : Z :=
  if zle i 1 then 1 else FAC (i - 1) * i.
```

FAC

implementation of factorial in Coq using `Function`
(recursion on `Acc` well-foundedness predicate):

```
Function FAC (i : Z) {measure Z.to_nat i} : Z :=
  if zle i 1 then 1 else FAC (i - 1) * i.
Proof. intros. apply Z2Nat.inj_lt; omega. Defined.
```

FAC

implementation of factorial in Coq using `Function`
(recursion on `Acc` well-foundedness predicate):

```
Function FAC (i : Z) {measure Z.to_nat i} : Z :=
  if zle i 1 then 1 else FAC (i - 1) * i.
Proof. intros. apply Z2Nat.inj_lt; omega. Defined.
```

and a trivial lemma that we will need later
(functions defined with `Function` do not reduce well):

```
Lemma FAC_step (i : Z) :
  i > 0 -> FAC (i + 1) = FAC i * (i + 1).
Proof.
intros. rewrite FAC_equation. destruct (zle (i + 1) 1).
omega. assert (i + 1 - 1 = i). omega. rewrite H0. auto.
Qed.
```

fac_spec

importing definitions from the generated `fac.v`:

```
Require Import fac.fac.
```

and the specification of our function:

```
Definition fac_spec :=
  DECLARE _fac
    WITH n : Z
    PRE [ _n OF tint ]
      PROP  (0 <= n <= Int.max_signed)
      LOCAL (temp _n (Vint (Int.repr n)))
      SEP   ()
    POST [ tint ]
      PROP  ()
      LOCAL (temp ret_temp (Vint (Int.repr (FAC n))))
      SEP   ().
```

PROP and LOCAL and SEP

assertions P consist of three parts:

- PROP
  does not refer to store or heap

$$(!!b)\ s\ h := \mathsf{emp}\ s\ h \wedge \mathsf{b}$$

- LOCAL
  refers only to store

$$(!b)\ s\ h := \mathsf{emp}\ s\ h \wedge \mathsf{eval}(b, s)$$

- SEP
  refers to both store and heap

$$(b)\ s\ h$$

`temp` and `data_at`

two basic assertions

- LOCAL assertion:

$$s(x) = n$$

$$\mathsf{temp}\ x\ n$$

- SEP assertion:

$$a_1 \mapsto a_2$$

$$\mathsf{data\_at}\ \pi\ \tau\ a_2\ a_1$$

τ = C type
π = permission

body_fac

'boilerplate' to define some variables related to `prog`:

```
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
Definition Gprog := augment_funspecs prog [fac_spec].
```

body_fac

'boilerplate' to define some variables related to prog:

```
Instance CompSpecs : compspecs. make_compspecs prog. Defined.
Definition Vprog : varspecs. mk_varspecs prog. Defined.
Definition Gprog := augment_funspecs prog [fac_spec].
```

correctness of the body of fac, using symbolic execution:

```
Lemma body_fac: semax_body Vprog Gprog f_fac fac_spec.
Proof.
start_function. forward. forward. forward.
forward_while (fac_inv n). Exists 1. entailer!. entailer!.
  forward. forward. forward. Exists (i + 1). entailer!.
  split. omega. rewrite FAC_step. auto. omega.
forward.
destruct H0. assert (i = n). omega. subst. entailer!.
destruct H0. subst. entailer!.
Qed.
```

all_funcs_correct

more 'boilerplate':

```
Existing Instance NullExtension.Espec.
```

including the correctness of the program
(combining the correctness of all functions):

```
Lemma all_funcs_correct :
  semax_func Vprog Gprog (prog_funct prog) Gprog.
Proof.
unfold Gprog, prog, prog_funct. semax_func_cons body_fac.
Qed.
```

fac_inv

the loop invariant that we had skipped
(this finishes the code from `verif_fac.v`):

```
Definition fac_inv (n : Z) : environ -> mpred :=
  EX i : Z,
    PROP  (1 <= i <= n \/ (n = 0 /\ i = 1))
    LOCAL (temp _n (Vint (Int.repr n));
           temp _i (Vint (Int.repr i));
           temp _f (Vint (Int.repr (FAC i))))
    SEP   ().
```

annotated program

$\{0 \leq n \leq 2^{31} - 1\}$
$i := 1;$
$f := 1;$
while $i < n$ do $\{(1 \leq i \leq n \vee (n = 0 \wedge i = 1)) \wedge f = i!\}$
  $i := i + 1;$
  $f := f \cdot i$
od
$\{f = n!\}$

annotated program

$\{0 \leq n \leq 2^{31} - 1\}$
$i := 1;$
$f := 1;$
while $i < n$ do $\{(1 \leq i \leq n \vee (n = 0 \wedge i = 1)) \wedge f = i!\}$
  $i := i + 1;$
  $f := f \cdot i$
od
$\{f = n!\}$

verification conditions:

$0 \leq n \leq 2^{31} - 1$
$\Rightarrow (1 \leq 1 \leq n \vee (n = 0 \wedge 1 = 1)) \wedge 1 = 1!$

$(1 \leq i \leq n \vee (n = 0 \wedge i = 1)) \wedge f = i! \wedge i < n$
$\Rightarrow (1 \leq i + 1 \leq n \vee (n = 0 \wedge i + 1 = 1)) \wedge f \cdot (i + 1) = (i + 1)!$

$(1 \leq i \leq n \vee (n = 0 \wedge i = 1)) \wedge f = i! \wedge \neg(i < n)$
$\Rightarrow f = n!$

DEMO

verifying SHA-256

sha.c in VST

counterpart of OpenSSL `crypto/sha/sha256.c`
247 lines of code, 1 typedef, 1 const array, 6 functions

```
typedef struct SHA256state_st {...} SHA256_CTX;
static const unsigned int K256[64] = ...;

void sha256_block_data_order(SHA256_CTX *ctx, const void *in)
void SHA256_addlength(SHA256_CTX *c, size_t len)
void SHA256_Init(SHA256_CTX *c)
void SHA256_Update(SHA256_CTX *c, const void *data_, size_t len)
void SHA256_Final(unsigned char *md, SHA256_CTX *c)
void SHA256(const unsigned char *d, size_t n, unsigned char *md)
```

fragment of `sha256_block_data_order`

after macro preprocessing:

```
...
for (i = 0; i < 16; i++) {
  (l = (((unsigned long) (*((data)++))) << 24), l |=
    (((unsigned long) (*((data)++))) << 16), l |=
    (((unsigned long) (*((data)++))) << 8), l |=
    (((unsigned long) (*((data)++)))), l);
  X[i] = l;
  Ki = K256[i];
  T1 = l + h + (((((e)) << (26)) | ((((e)) & 0xffffffff) >> (32 - (26)))) ^
    ((((e)) << (21)) | ((((e)) & 0xffffffff) >> (32 - (21)))) ^
    ((((e)) << (7)) | ((((e)) & 0xffffffff) >> (32 - (7))))) +
    (((e) & (f)) ^ ((~(e)) & (g))) + Ki;
  T2 = (((((a)) << (30)) | ((((a)) & 0xffffffff) >> (32 - (30)))) ^
    ((((a)) << (19)) | ((((a)) & 0xffffffff) >> (32 - (19)))) ^
    ((((a)) << (10)) | ((((a)) & 0xffffffff) >> (32 - (10))))) +
    (((a) & (b)) ^ ((a) & (c)) ^ ((b) & (c)));
  h = g; g = f; f = e; e = d + T1;
  d = c; c = b; b = a; a = T1 + T2;
  }
...
```

an array on the stack

another fragment of `sha256_block_data_order`:

```
#define SHA_LONG unsigned int

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;
    ...
```

some other frameworks cannot handle local array variables!

fac versus sha

- counterpart of `fac.c` is `sha.c`
- counterpart of `fac.v` is `sha.v`
- counterparts of `verif_fac.v`:
  - counterpart of `FAC` is the file `SHA256.v` with:
    - `SHA_256`
  - counterpart of `fac_spec` is the file `spec_sha.v` with:
    - `sha256_block_data_order_spec`
    - `SHA256_addlength_spec`
    - `SHA256_Init_spec`
    - `SHA256_Update_spec`
    - `SHA256_Final_spec`
    - `SHA256_spec`
  - counterparts of `body_fac` are the files:
    - `verif_sha_bdo.v` with `body_sha256_block_data_order`
    - `verif_addlength.v` with `body_SHA256_addlength`
    - `verif_sha_init.v` with `body_SHA256_Init`
    - `verif_sha_update.v` with `body_SHA256_Update`
    - `verif_sha_final.v` with `body_SHA256_Final`
    - `verif_SHA256.v` with `body_SHA256`
  - *plus several other files with lots of lemmas*

two executable versions

Fibonacci function with naive recursion takes exponential time

two executable versions

Fibonacci function with naive recursion takes exponential time

- `SHA256.v` defines `SHA256`

  naive implementation
  exponential time

- `functional_prog.v` defines `SHA256'`

  better implementation
  reasonable time
  about a million times slower than C implementation

- ... and proves `SHA256' = SHA256`

testing on examples: seem okay

sizes and times

lines	seconds	component
842	17	lemmas about the functional spec
756	27	lemmas about the API spec
1613	47	verification of `sha256_block_data_order`
251	61	verification of `SHA256_addlength`
34	102	verification of `SHA256_Init`
1745	658	verification of `SHA256_Update`
1276	781	verification of `SHA256_Final`
37	38	verification of `SHA256`
6555	2358	total

trusting the spec

why trust that the spec is correct?

- C program: `sha.c`
  268 lines
- Coq specification: `SHA256.v` + `spec_sha.v`
  154 + 210 = 364 lines

trust

trusting the spec

why trust that the spec is correct?

- C program: `sha.c`
  268 lines
- Coq specification: `SHA256.v` + `spec_sha.v`
  154 + 210 = 364 lines

possible to prove crypto properties of the Coq definitions

is it really OpenSSL?

changes in the code:

- macros expanded to the SHA-256 case
- compiled to Clight in a specific way
- adapted: no side effects inside subexpressions
- adapted: no memory references inside subexpressions
- some additional return statements

current version of OpenSSL not close to this any more

trusted computing base

VST framework

- CompCert semantics of C
  'is it really C?'
  (**not** part of TCB when compiling using CompCert)
- CompCert semantics of assembly
- Calculus of Inductive Constructions
- source code of Coq **kernel**
- source code of OCaml compiler and runtime
- microprocessor

SHA correctness

- SHA specification
- C compiler
  (**not** part of TCB when compiling using CompCert)
- assembler
- microprocessor

axioms

```
Print Assumptions all_funcs_correct.
```

axioms

```
Print Assumptions all_funcs_correct.
```

- Classical_Prop.classic :
  ```
  forall P : Prop, P \/ ~ P
  ```
- prop_ext :
  ```
  forall A B : Prop, (A <-> B) -> A = B
  ```
- functional_extensionality_dep :
  ```
  forall (A : Type) (B : A -> Type)
    (f g : forall x : A, B x),
  (forall x : A, f x = g x) -> f = g
  ```

axioms

```
Print Assumptions all_funcs_correct.
```

- `Classical_Prop.classic :`
  `forall P : Prop, P \/ ~ P`
- `prop_ext :`
  `forall A B : Prop, (A <-> B) -> A = B`
- `functional_extensionality_dep :`
  ```
  forall (A : Type) (B : A -> Type)
    (f g : forall x : A, B x),
  (forall x : A, f x = g x) -> f = g
  ```
- **26 axioms** about real numbers
  through CompCert (floating point) through Floc

axioms

```
Print Assumptions all_funcs_correct.
```

- `Classical_Prop.classic` :
  `forall P : Prop, P \/ ~ P`
- `prop_ext` :
  `forall A B : Prop, (A <-> B) -> A = B`
- `functional_extensionality_dep` :
  ```
  forall (A : Type) (B : A -> Type)
    (f g : forall x : A, B x),
  (forall x : A, f x = g x) -> f = g
  ```
- **26 axioms** about real numbers
  through CompCert (floating point) through Floc
- **19 axioms** about `semax`
  proved in `veric/SeparationLogicSoundness.v`
  ('I didn't want to complicate things with a Functor application somewhere')

$17! = -288522240$?

%

$17! = -288522240$?

```
% factest
```

$17! = -288522240$?

```
% factest
17
```

$17! = -288522240$?

```
% factest
17
-288522240
%
```

$17! = -288522240$?

```
% factest
17
-288522240
%
```

- overflow in C11 standard: *undefined behavior* = crash
  (computer can do whatever it likes)
- overflow in CompCert: wraps mod 2^{32}
  (in that case: the computer likes to wrap ☺)
- **but didn't we prove that the program calculates $n!$?**
  (where is the 'modulo' in the specification?)

$17! = -288522240$?

```
% factest
17
-288522240
%
```

- overflow in C11 standard: *undefined behavior* = crash
  (computer can do whatever it likes)
- overflow in CompCert: wraps mod 2^{32}
  (in that case: the computer likes to wrap ☺)
- **but didn't we prove that the program calculates $n!$?**
  (where is the 'modulo' in the specification?)

$$s(x) = n$$

`temp` x `(Vint (Int.repr` n`))`

$17! = -288522240$?

```
% factest
17
-288522240
%
```

- overflow in C11 standard: *undefined behavior* = crash
  (computer can do whatever it likes)
- overflow in CompCert: wraps mod 2^{32}
  (in that case: the computer likes to wrap ☺)
- **but didn't we prove that the program calculates $n!$?**
  (where is the 'modulo' in the specification?)

$$s(x) = n$$

`temp` x `(Vint (Int.repr` n`))`

not: 'the value of x is n'
but: 'the value of x is the residue of n mod 2^{32}'

related work

specification, implementation, foundational, automatic, general

- **specification**
  is there a specification of the program's function?
  amenable to analysis in a proof assistant?
- **implementation**
  is the proof about an efficient implementation?
- **foundational**
  is there an end-to-end machine-checked proof from the foundations of logic?
- **automatic** (VST: ☹)
  check or synthesize the program without much interactive human input or annotations?
- **general**
  can the verifier handle all parts of the program?

larger C verification projects

two recent large C verification projects

- **seL4 in Isabelle**
  Gerwin Klein et al. 2009
  NICTA, Australia

- **CertiKOS in Coq**
  Liang Gu et al. 2011
  Yale, US

larger C verification projects

two recent large C verification projects

- **seL4 in Isabelle**
  Gerwin Klein et al. 2009
  NICTA, Australia

- **CertiKOS in Coq**
  Liang Gu et al. 2011
  Yale, US

but:

- no separation logic
- no function pointers/higher order specifications
- seL4 and CertKOS: newly written code
- no arrays on the stack

who has choices need not choose

different approaches to program verification:

- static analysis
- model checking
- **interactive theorem provers**
  type theory!

table of contents

contents