First Building Blocks For Implementations of Security Protocols Verified in Coq

Reynald Affeldt[1)] Kazuhiko Sakaguchi[1)2)]

[1)] National Institute of Advanced Industrial Science and Technology, Japan
[2)] University of Tsukuba

Motivation

- Long-term goal:
  - Verified implementation of a security protocol in Coq
- Results so far:
  - Important pieces of assembly and C code
    - Progress reports in other venues [SAC 2012, PLPV 2013]
    - Recently completed
- Why this presentation?
  - Much related work in verification of low-level code
  - Not that many examples of concrete pieces of code
  - Significant effort worth reusing

Concrete Verification Targets

- Pieces of code typical of security protocols
  - E.g., consider the SSL/TLS protocol:
    - Core = cryptographic schemes
      - Partly implemented in assembly
        - » Performance, security counter-measures
      - Mostly modular arithmetic:
        - » Modular exponentiation (e.g., all steps of ElGamal)
        - » Pseudo-random number generation (key generation, probabilistic encryption)
        - » Extended GCD algorithm (e.g., inverse modulo for private keys of RSA)
    - Communication = exchange of formatted binary packets
      - Parsing/pretty-printing
      - Usually implemented in C

Previous work

This talk

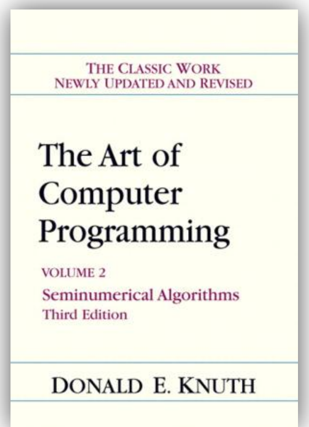
Outline

- Formal verification of arithmetic functions
  - Case study: binary extended GCD
- Formal verification of binary packet parsing
  - Case study: parsing of initialization packets for TLS
- Related work and conclusion

Binary Extended GCD

Algorithm in Pseudo-code

- Extended? Given u and v, return $u * u_1 + v * u_2 = g * u_3 = GCD(u,v)$
- Binary? Multi-precision division → shifts
- Knuth's binary extended GCD ≈ 49 lines




```
Definition begcd g u v u1 u2 u3 v1 v2 v3 t1 t2 t3 :=
  g ← 1 ;
  prelude u v g ;
  init u v u1 u2 u3 v1 v2 v3 t1 t2 t3 ;
  WHILE t3 ≠ 0 {
    WHILE t3 % 2 = 0 { halve u v t1 t2 t3 } ;
    reset u v u1 u2 u3 v1 v2 v3 t1 t2 t3 ;
    subtract u v u1 u2 u3 v1 v2 v3 t1 t2 t3 }.
```

```
Definition prelude x y g :=
  WHILE x % 2 = 0 && y % 2 = 0 {
    x ← x / 2 ;
    y ← y / 2 ;
    g ← g × 2 }.
```

```
Definition init
    u v u1 u2 u3 v1 v2 v3 t1 t2 t3 :=
  u1 ← 1 ;
  u2 ← 0 ;
  u3 ← u ;
  v1 ← v ;
  v2 ← 1 − u ;
  v3 ← v ;
  IF u % 2 = 1 THEN
    t1 ← 0 ;
    t2 ← −1 ;
    t3 ← − v
  ELSE
    t1 ← 1 ;
    t2 ← 0 ;
    t3 ← u.
```

```
Definition subtract
    u v u1 u2 u3 v1 v2 v3 t1 t2 t3 :=
  t1 ← u1 − v1 ;
  t2 ← u2 − v2 ;
  t3 ← u3 − v3 ;
  IF 0 ≥ t1 THEN
    t1 ← t1 + v ;
    t2 ← t2 − u
  ELSE
    skip.
```

```
Definition reset
    u v u1 u2 u3 v1 v2 v3 t1 t2 t3 :=
  IF t3 ≥ 0 THEN
    u1 ← t1 ;
    u2 ← t2 ;
    u3 ← t3
  ELSE
    v1 ← v − t1 ;
    v2 ← − (u + t2) ;
    v3 ← − t3.
```

```
Definition halve u v t1 t2 t3 :=
  IF t1 % 2 = 0 && t2 % 2 = 0 THEN
    t1 ← t1 / 2 ;
    t2 ← t2 / 2 ;
    t3 ← t3 / 2
  ELSE
    t1 ← (t1 + v) / 2 ;
    t2 ← (t2 − u) / 2 ;
    t3 ← t3 / 2.
```

Binary Extended GCD

From Pseudo-code to Assembly

```
Definition begcd g u v u1 u2 u3 v1 v2 v3 t1 t2 t3 :=
  g ← 1 ;
  prelude u v g ;
  init u v u1 u2 u3 v1 v2 v3 t1 t2 t3 ;
  WHILE t3 ≠ 0 {
    WHILE t3 % 2 = 0 { halve u v t1 t2 t3 } ;
    reset u v u1 u2 u3 v1 v2 v3 t1 t2 t3 ;
    subtract u v u1 u2 u3 v1 v2 v3 t1 t2 t3 }.
```

```
Definition begcd_mips rk rg ru rv ru1 ru2 ru3
 rv1 rv2 rv3 rt1 rt2 rt3 a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 :=
 multi_one_u rk rg a0 a1 ;
 prelude_mips rk rg ru rv a0 a1 a2 a3 ;
 init_mips rk ru rv ru1 ru2 ru3 rv1 rv2 rv3
           rt1 rt2 rt3 a0 a1 a2 a3 a4 a5 a6 ;
 pick_sign rt3 a0 a1 ;
 WHILE (bne a1 r0) {
  multi_is_even_s rt3 a0 a1 a2 ;
  WHILE (bne a2 r0) {
   halve_mips rk ru rv rt1 rt2 rt3
              a0 a1 a2 a3 a4 a5 a6;
   multi_is_even_s rt3 a0 a1 a2 } ;
  reset_mips rk ru rv ru1 ru2 ru3 rv1 rv2 rv3
             rt1 rt2 rt3 a0 a1 a2 a3 a4 a7 a8 a9 ;
  subtract_mips rk ru rv ru1 ru2 ru3 rv1 rv2 rv3
                rt1 rt2 rt3 a0 a1 a2 a3 a4 a5 a6 a7 a8 ;
  pick_sign rt3 a0 a1 }.
```

(69 l.o.c of MIPS)

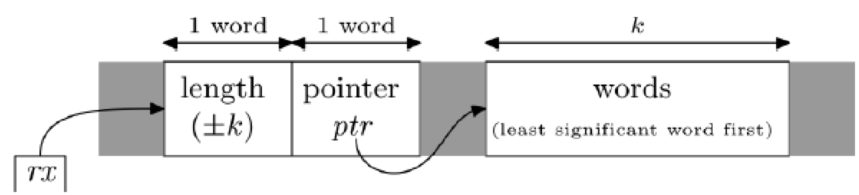
Main issue:

Arbitrary-size integers → Multi-precision integers
(In other words, quid of overflows?)

> *"in many cases the intellectual heart of a program lies in the ingenious choice of data representation rather than in the abstract algorithm" (J.C. Reynolds, 1981)*

Starting point:

Signed integers like in the celebrated GMP library

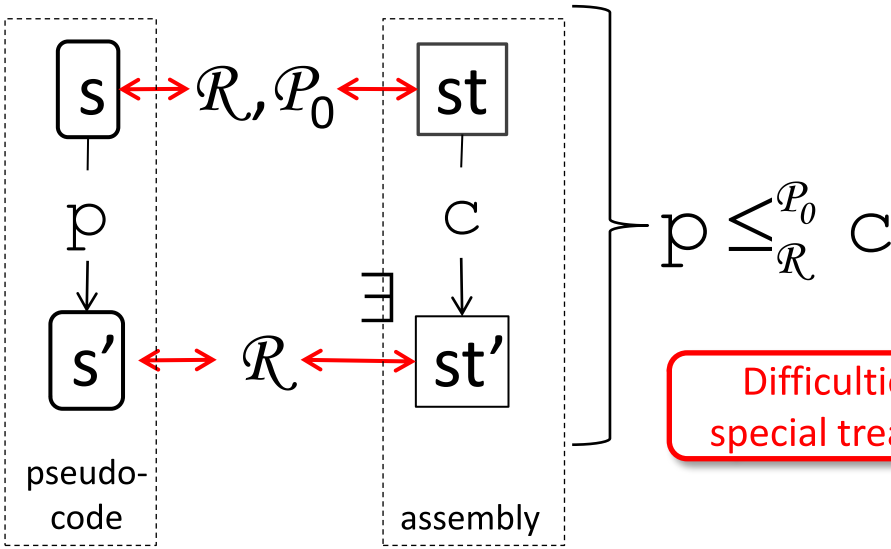


Library of verified arithmetic functions:

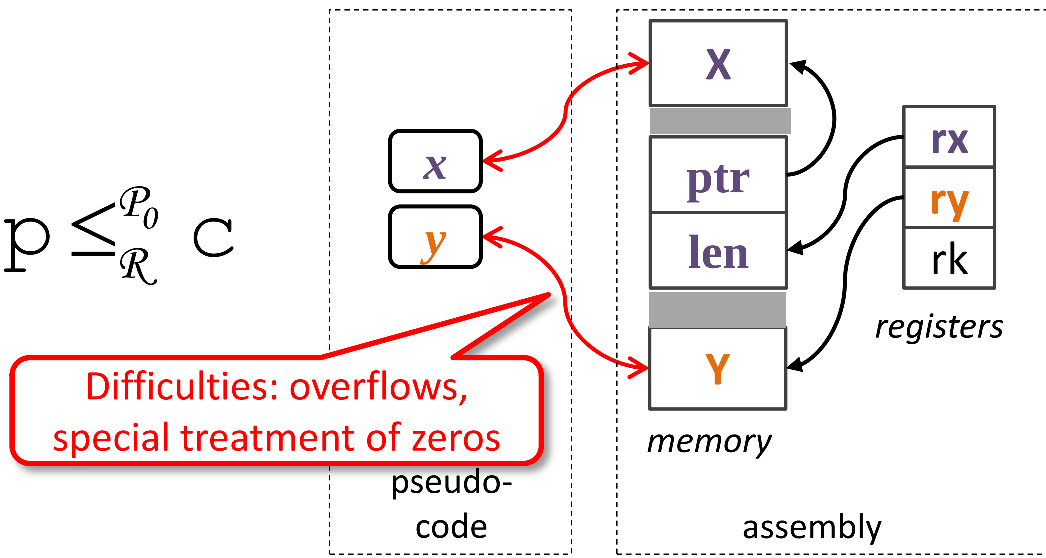
Signed additions, subtraction, halving, doubling, etc. (25 functions, 313 l.o.c. of MIPS)

Pseudo-code ↔ Assembly

- Forward simulation:



- $\mathcal{R}$ for arithmetic (e.g.):



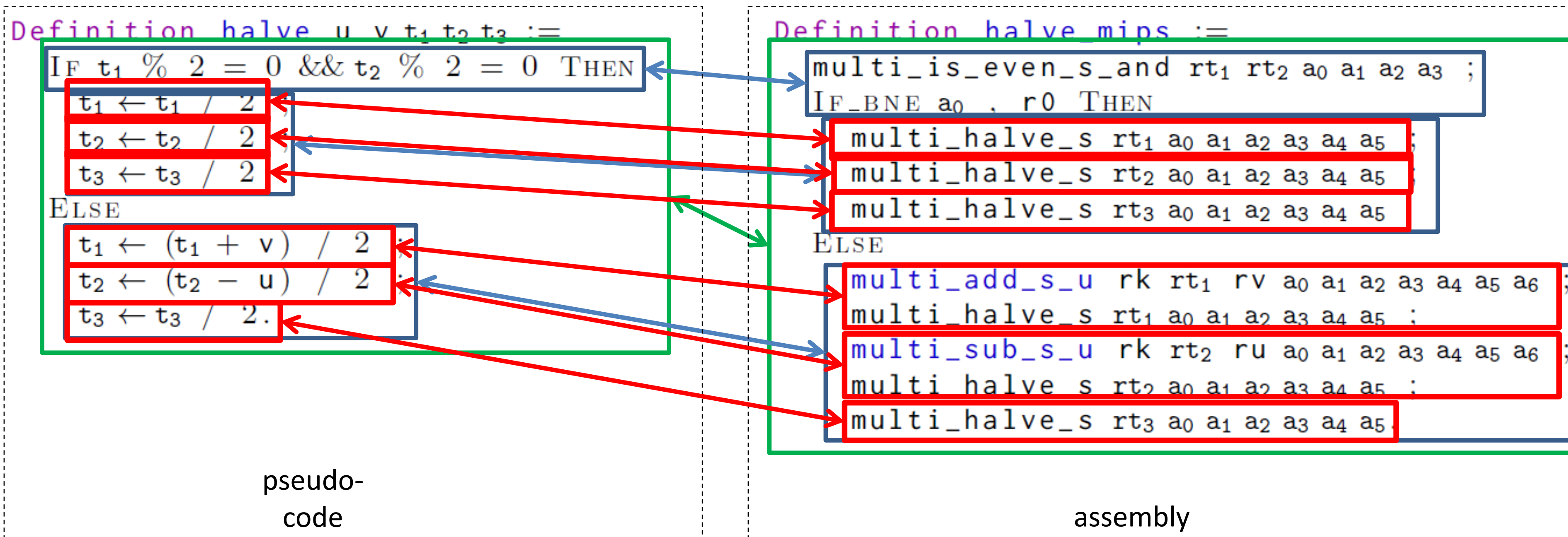
- Compositional reasoning (e.g.):

$$\frac{p \leq^{\mathcal{P}}_{\mathcal{R}} c \qquad p' \leq^{\mathcal{Q}}_{\mathcal{R}} c'}{p;p' \leq^{\mathcal{P}}_{\mathcal{R}} c;c'} \; [\mathcal{P}]p \downarrow c[\mathcal{Q}]$$

Pseudo-code ↔ Assembly

Simulation Proof

1. Decompose using compositional reasoning
2. Basic simulations proved using ***support library***



<u>Example:</u> One of the five steps of the binary extended gcd

Binary Extended GCD in Assembly
Technical Verification Overview

- Support library
  - Verification of basic functions for ***signed*** multi-precision arithmetic
    - Signed additions, substractions, halving, doubling, etc. (25 functions, 313 l.o.c. of MIPS)
    - Prove correctness (7,746 l.o.c. of Coq scripts)
    - Simulation statements (4,753 l.o.c. of Coq scripts)
- Application to Knuth's binary extended GCD
  1. Formal verification of the pseudo-code
     - Loop-invariants about functional correctness
  2. 1,466 l.o.c of ***systematic*** Coq scripts (for 69 l.o.c. of MIPS)
     - Invariants about implementation details only (overflows)
- Details:
  - [On Construction of A Library of Formally Verified Low-level Arithmetic Functions, ISSE 9(2): 59-77 (2013)]

Outline

- Formal verification of arithmetic functions
  - Case study: binary extended GCD
- Formal verification of binary packet parsing
  - Case study: parsing of initialization packets for TLS
- Related work and conclusion

An Intrinsic Encoding of a subset of C

- Expressions indexed with (type-checking rules for) C types:

Inductive exp {g σ} : g.-typ → Type

Variable | var_e : ∀ str t, get str σ = ⌊ t ⌋ → exp t

Constant | cst_e : ∀ t, t.-phy → exp t

Arithmetic addition | add_e : ∀ t, exp (btyp: t) → exp (btyp: t) → exp (btyp: t)

Pointer arithmetic | add_p : ∀ t, exp (:* t) → exp (btyp: sint) → exp (:* t)

same Notation “a ¥+ b” := … using Class/Instance

- Usefulness:

$[\ 1\ ]_{sc}$: exp (btyp: sint)

%”buf” : exp (:* (btyp: uchar))

Arithmetic addition: $[\ 1\ ]_{sc} + [\ 1\ ]_{sc}$ ○

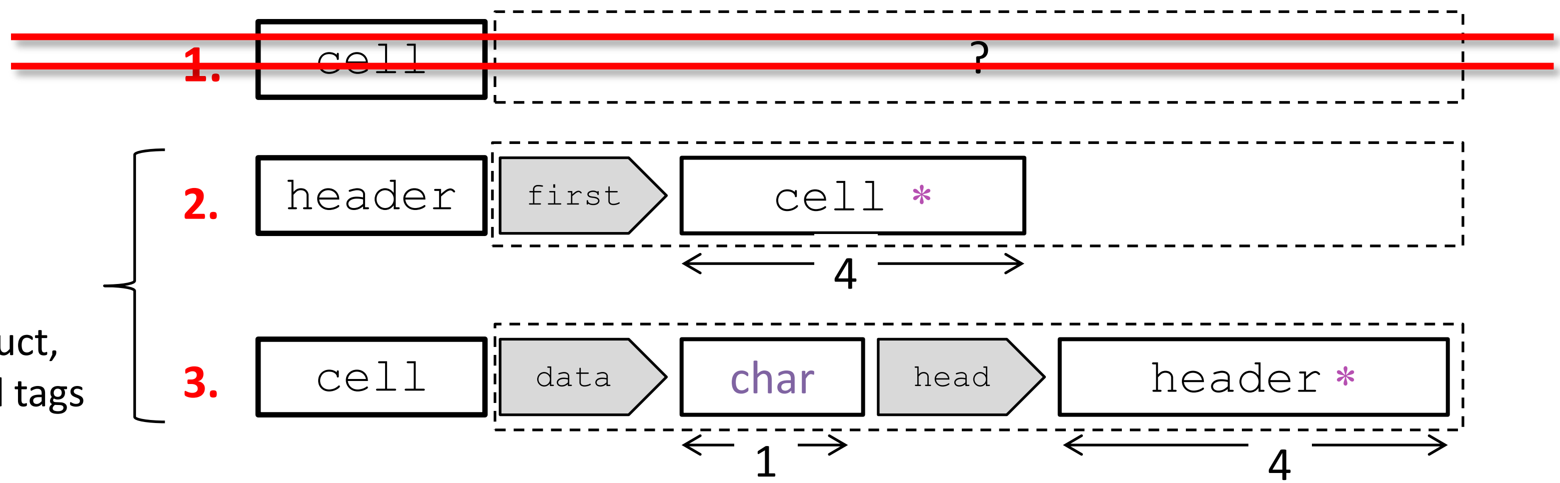
Pointer arithmetic: %”buf” + $[\ 1\ ]_{sc}$ ○

%”buf” + %”buf” ⦸

Deep embedding of C Types

- Example of a C structure:

```
1. {struct cell ;
2.  struct header {struct cell *first;};
3.  struct cell   {char data; struct header *head;};}
```



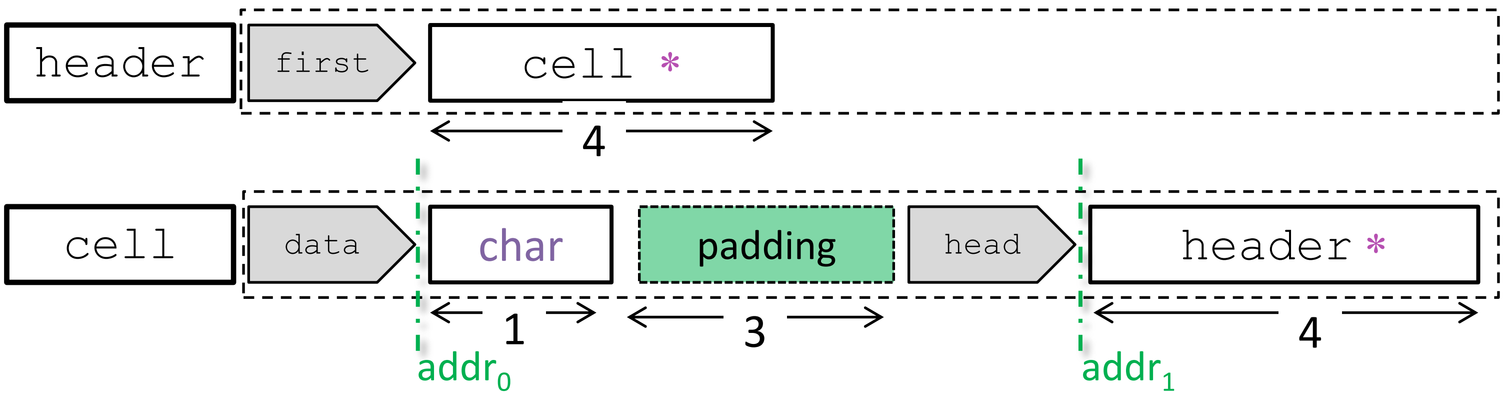
Valid structure:
No cycle,
no empty struct,
no undefined tags

Generic terminating type traversal function:

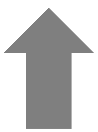
```
Program Definition typ_traversal (ty : g.-typ) : Res :=

Record config {Res Accu : Type} := mkConfig {
  f_ityp : ityp -> Res ;
  f_ptyp : typ -> Res ;
  f_styp_iter : Accu -> string * g.-typ * Res -> Accu ;
  f_styp_fin : tag * g.-typ -> (Accu -> Accu) -> Res ;
  f_atyp : nat -> tag * g.-typ -> Res -> Res }.
```

Application to sizeof Computation

- C structures are padded to conform to alignment:




```
Goal (sizeof cell = 1 + 3 + 4). by []. Qed.
```



Obtained by instantiating of the generic type traversal:

```
Definition sizeof_config g := mkConfig g
  sizeof_ityp
  (fun _ => sizeof_ptr)
  (fun a x => a + padd a (align x.1.2) + x.2)
  (fun ty a => a 0 + padd (a 0) (align ty.2))
  (fun s _ r => muln s r).
```

Application to Pretty-printing (new)

- Pretty-printer = instantiation of the generic type traversal:

```
Definition pp_config {g} := (mkConfig g
  (fun t name tl => ityp_to_string t (" " ++ name ++ tl))
  (fun t name tl => typ_to_string t ("(*" ++ name ++ ")") tl)
  (fun accu p => accu ++ p.2 p.1.1 ("; "))
  (fun p f name tl => "struct " ++
    struct_tag_to_string p.1 (" { " ++ f "" ++ "} " ++ name ++ tl))
    (fun sz _ f name tl => f name ("[" ++ pp_nat sz ("]" ++ tl))))%string.
```

- Example:

```
{struct cell ;
 struct header {struct cell *first;};
 struct cell   {char data; struct header *head;};}
```

```
Goal PrintAxiom _ (typ_to_string_rec g cell "" "").
compute.

============================
 PrintAxiom string
   "struct cell { unsigned char data; struct header (*head); } "
```

Case Study (1/2)

Parsing of Network Packets for SSL/TLS

Coq model

```
Definition ssl_parse_client_hello1 cont :=
  _ret <-ssl_fetch_input(__ssl, [ 5 ]sc) ;
  If b[ __ret \!= [ 0 ]sc ] Then
    ret
  Else (
  _buf <-* __ssl .=> get_in_hdr ;
  _buf0 <-* __buf ;
  If b[ (__buf0 \& [ 128 ]8uc) \!= [ 0 ]8uc ] Then
    _ret <- [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c;
    ret
  Else
  If b[ __buf0 \!= [ SSL_MSG_HANDSHAKE ]c ] Then
    _ret <- [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ;
    ret
  Else (
  _buf1 <-* __buf \+ [ 1 ]sc ;
  If b[ __buf1 \!= [ SSL_MAJOR_VERSION_3 ]c ] Then
    _ret <- [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ;
    ret
  Else (
  _buf3 <-* __buf \+ [ 3 ]sc ;
  _buf4 <-* __buf \+ [ 4 ]sc ;
  _n <- (( (int) __buf3) \<<e [ 8 ]sc) \| (int) __buf4 ;
  If b[ __n \<e [ 45 ]sc ] Then
    _ret <- [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c;
    ret
  Else (
  If b[ __n \>e [ 512 ]sc ] Then
    _ret <- [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c ;
    ret
  Else (
```

PolarSSL (polarssl.org)

```
static int ssl_parse_client_hello( ssl_context *ssl )
{
    int ret, i, j, n;
    int ciph_len, sess_len;
    int chal_len, comp_len;
    unsigned char *buf, *p;

    SSL_DEBUG_MSG( 2, ( "=> parse client hello" ) );

    if( ( ret = ssl_fetch_input( ssl, 5 ) ) != 0 )
    {
        SSL_DEBUG_RET( 1, "ssl_fetch_input", ret );
        return( ret );
    }

    buf = ssl->in_hdr;

    if( ( buf[0] & 0x80 ) != 0 )
    {
        SSL_DEBUG_BUF( 4, "record header", buf, 5 );

        SSL_DEBUG_MSG( 3, ( "client hello v2, message type: %d",
                       buf[2] ) );
        SSL_DEBUG_MSG( 3, ( "client hello v2, message len.: %d",
                       ( ( buf[0] & 0x7F ) << 8 ) | buf[1] ) );
        SSL_DEBUG_MSG( 3, ( "client hello v2, max. version: [%d:%d]",
                       buf[3], buf[4] ) );

        /*
         * SSLv2 Client Hello
```

Concrete C Syntax

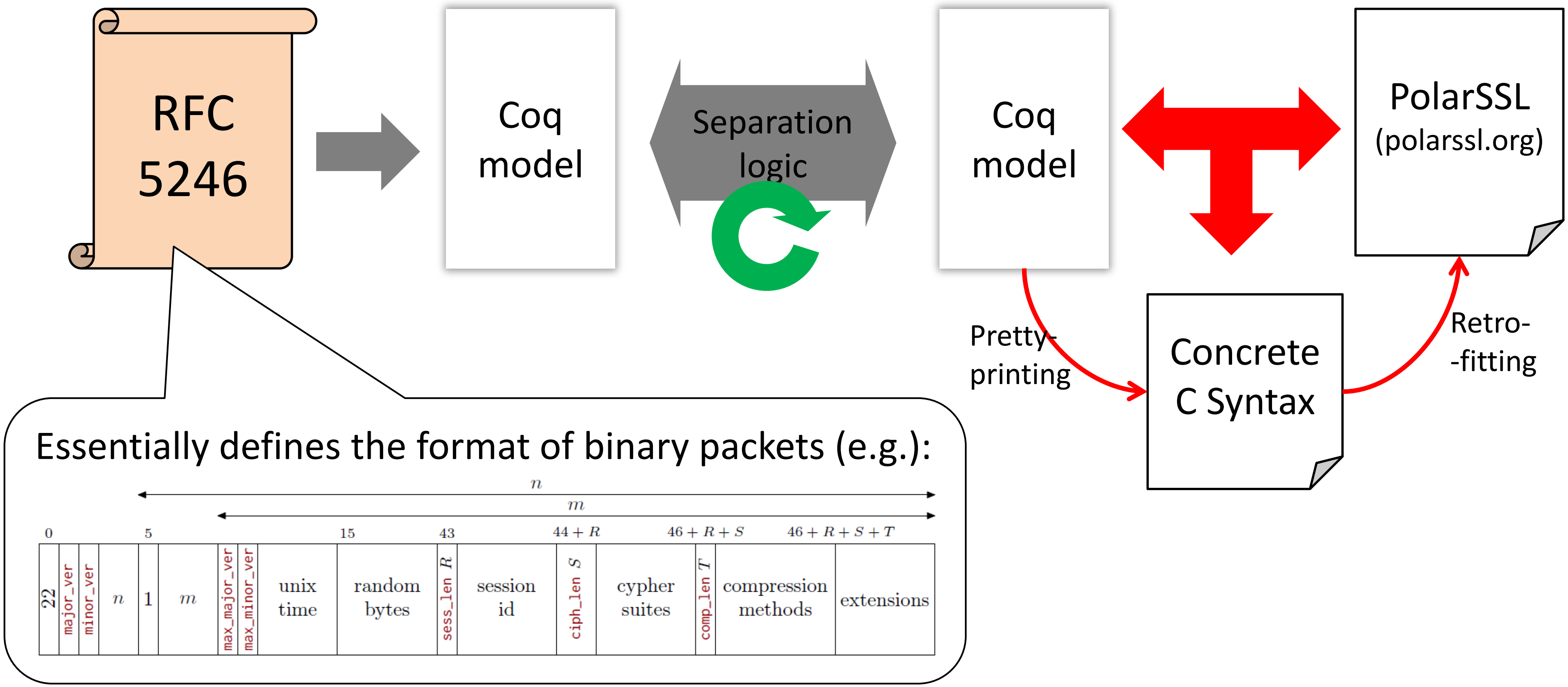
```
"ret = ssl_fetch_input(ssl, 5);
if (((ret) != (0))) {
;
} else {
buf = *(ssl)->in_hdr;
_buf0_ = *buf;
if ((((_buf0_) & (128u)) != (0u))) {
ret = -38912;
;
} else {
if (((_buf0_) != (22u))) {
ret = -38912;
;
} else {
_buf1_ = *(buf) + (1);
if (((_buf1_) != (3u))) {
ret = -38912;
;
} else {
_buf3_ = *(buf) + (3);
_buf4_ = *(buf) + (4);
n = (((unsigned char)(_buf3_)) << (8)) | ((unsigned char)(_buf4_));
if (((n) < (45))) {
ret = -38912;
;
} else {
if (((n) > (512))) {
ret = -38912;
;
} else {
```

Pretty-printing

Retrofitting

Case Study (2/2)

Parsing of Network Packets for SSL/TLS



ClientHello Parsing (1/2)

Technical Verification Overview

- Target function: `ssl_parse_client_hello`
  - Original C code: 161 l.o.c. (85 w.o. comments and debug info)
  - Coq model: 132 l.o.c. (Patched version!)
    - goto → while
    - Expressions with side-effects → split into commands
- Formal proof:
  - 4087 l.o.c. (≈ 30 l.o.c. Coq scripts / l.o.c. of C)
  - Ltac tactics (a la Appel [2006])
  - Low-level manipulation of bit strings (shifts, concats, etc.) and overflow checking occupy much space
- Benefits of formal verification:
  - Debugging of the original C code:
    - To prevent accesses to allocated but not initialized memory
    - To guarantee conformance to RFC
      - Check for the absence of *extensions*
  - Restrictions w.r.t. RFC have been made explicit
    - Some features are not implemented (by design?), but which ones?

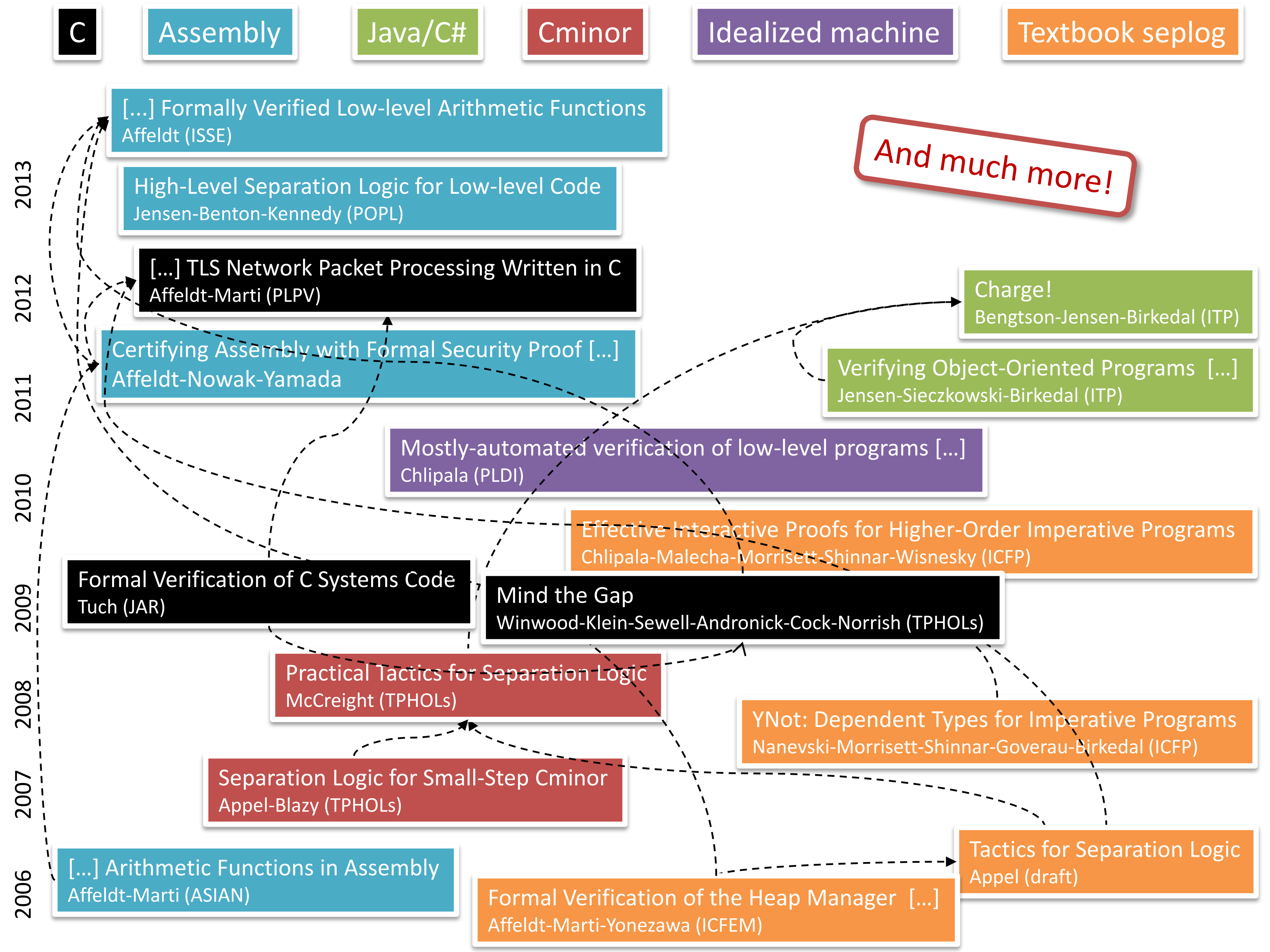
ClientHello Parsing (2/2)

Technical Verification Overview

- Compilation of `ssl_parse_client_hello`'s proof:
  - ≈ 220 min. (Unix time)
  - ≈ 9 GB of RAM
- Bottleneck:
  - Most time spent checking a nested loop (for cipher search)
    - Where Separation logic assertions are large because of invariants
- Counter-measures:
  - Hide string constants behind identifiers
  - Careful management of hypotheses
  - Rewrite Program functions by hand
    - `lazy` rather than `compute`
  - Ad-hoc lemmas rather than Ltac tactics
    - Trade-off short scripts ↔ compilation/maintenance time

Outline

- Formal verification of arithmetic functions
  - Case study: binary extended GCD
- Formal verification of binary packet parsing
  - Case study: parsing of initialization packets for TLS
- Related work and conclusion



C
Assembly
Java/C#
Cminor
Idealized machine
Textbook seplog
And much more!
2013
2012
2011
2010
2009
2008
2007
2006
[...] Formally Verified Low-level Arithmetic Functions
Affeldt (ISSE)
High-Level Separation Logic for Low-level Code
Jensen-Benton-Kennedy (POPL)
[...] TLS Network Packet Processing Written in C
Affeldt-Marti (PLPV)
Charge!
Bengtson-Jensen-Birkedal (ITP)
Certifying Assembly with Formal Security Proof [...]
Affeldt-Nowak-Yamada
Verifying Object-Oriented Programs [...]
Jensen-Sieczkowski-Birkedal (ITP)
Mostly-automated verification of low-level programs [...]
Chlipala (PLDI)
Effective Interactive Proofs for Higher-Order Imperative Programs
Chlipala-Malecha-Morrisett-Shinnar-Wisnesky (ICFP)
Formal Verification of C Systems Code
Tuch (JAR)
Mind the Gap
Winwood-Klein-Sewell-Andronick-Cock-Norrish (TPHOLs)
Practical Tactics for Separation Logic
McCreight (TPHOLs)
YNot: Dependent Types for Imperative Programs
Nanevski-Morrisett-Shinnar-Goverau-Birkedal (ICFP)
Separation Logic for Small-Step Cminor
Appel-Blazy (TPHOLs)
Tactics for Separation Logic
Appel (draft)
[...] Arithmetic Functions in Assembly
Affeldt-Marti (ASIAN)
Formal Verification of the Heap Manager [...]
Affeldt-Marti-Yonezawa (ICFEM)

Conclusion

- Summary:
  - Formal verification of concrete pieces of low-level code
    - Arithmetic functions in assembly
    - Network packet processing in C

  $\Rightarrow$ Our work provides concrete clues about the verification of security protocols in Coq
- Development tarballs online :
  - http://staff.aist.go.jp/reynald.affeldt/**coqdev**
- Future work:
  - Enable verification of program mixing assembly and C