First Building Blocks For Implementations of Security Protocols Verified in Coq

Reynald Affeldt\textsuperscript{1)} Kazuhiko Sakaguchi\textsuperscript{1,2)}

\textsuperscript{1)} National Institute of Advanced Industrial Science and Technology, Japan
\textsuperscript{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)
  • **Communication** = exchange of formatted binary packets
    – Parsing/pretty-printing
    – Usually implemented in C
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 \( u \times u_1 + v \times u_2 = g \times u_3 = \text{GCD}(u,v) \)
- Binary? Multi-precision division \(\rightarrow\) shifts
- Knuth’s binary extended GCD \(\approx 49\) lines
Binary Extended GCD

From Pseudo-code to Assembly

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:

\[ \begin{align*}
\text{p} & \leq_{\mathcal{R}} \text{c} \\
\text{p}' & \leq_{\mathcal{R}} \text{c}' \\
\text{p};\text{p}' & \leq_{\mathcal{R}} \text{c};\text{c}' \\
\end{align*} \]

\[ [\mathcal{P}]_{\text{p}} \downarrow \text{c}[\mathcal{Q}] \]

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

- Compositional reasoning (e.g.):

\[ \begin{align*}
\text{s} & \xrightarrow{\mathcal{R}, \mathcal{P}_0} \text{st} \\
\text{p} & \xrightarrow{\mathcal{R}} \text{c} \\
\text{s}' & \xleftarrow{\mathcal{R}} \text{st}' \\
\end{align*} \]
Pseudo-code ↔ Assembly

Simulation Proof

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

Example: 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 \textit{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 \textit{systematic} Coq scripts (for 69 l.o.c. of MIPS)
     • Invariants about implementation details only (overflows)

• Details:
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 expr \( \{g \sigma\} : g.-typ \rightarrow Type \)

Variable

| var_e : \( \forall str t, \text{get str } \sigma = \lfloor t \rfloor \rightarrow \text{exp } t \)

Constant

| cst_e : \( \forall t, t.-phy \rightarrow \text{exp } t \)

Arithmetic addition

| add_e : \( \forall t, \text{exp (btyp: } t) \rightarrow \text{exp (btyp: } t) \rightarrow \text{exp (btyp: } t) \)

| add_p : \( \forall t, \text{exp (}\ast t) \rightarrow \text{exp (btyp: sint) } \rightarrow \text{exp (}\ast t) \)

• Usefulness:

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

| \%”buf” : \text{exp (}\ast (\text{btyp: uchar}) \)

Arithmetic addition:

| \([1]_{sc} + [1]_{sc} \)

Pointer arithmetic:

| \%”buf” + [1]_{sc} \)

| \%”buf” + \%”buf” \)

same

Notation “a ¥+ b” := ...

using

Class/Instance
Deep embedding of C Types

- Example of a C structure:

```c
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:

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

Record config {Res Accu : Type} := mkConfig {
  f_i typ : ityp -> Res ;
  f_p typ : typ -> Res ;
  f_s typ _iter : Accu -> string * g.-typ * Res -> Accu ;
  f_s typ _fin : tag * g.-typ -> (Accu -> Accu) -> Res ;
  f_a typ : nat -> tag * g.-typ -> Res -> Res }
```
Application to sizeof Computation

- C structures are padded to conform to alignment:

```
header first cell *
```

```
cell data char padding head header *
```

Goal \( \text{sizeof cell} = 1 + 3 + 4 \). by \([\text{.}]\). 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

Definition: `ssl_parse_client_hello` 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]Buc) !=[0]Buc ] 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) <= [8]sc) \ (Int) _buf4 ;
    If b[ _n <= [45]sc ] Then
      _ret <- [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c;
    ret Else 
    If b[ _n >= [512]sc ] Then
      _ret <- [ POLARSSL_ERR_SSL_BAD_HS_CLIENT_HELLO ]c;
    ret Else 
  )
)

Coq model

PolarSSL (polarssl.org)

Pretty-printing

Concrete C Syntax

Retrofitting
Case Study (2/2)
Parsing of Network Packets for SSL/TLS

Essentially defines the format of binary packets (e.g.):

<table>
<thead>
<tr>
<th>0</th>
<th>5</th>
<th>15</th>
<th>43</th>
<th>44 + R</th>
<th>46 + R + S</th>
<th>46 + R + S + T</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>max_major_ver</td>
<td>max_minor_ver</td>
<td>unix time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>random bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>session id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cypher suites</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>compression methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>extensions</td>
</tr>
</tbody>
</table>

RFCC 5246
Coq model
Separation logic
Coq model
PolarSSL (polarssl.org)
Concrete C Syntax
Pretty-printing
Retro-fitting
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:
  – \( \approx 220 \) min. (Unix time)
  – \( \approx 9 \) 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 \( \leftrightarrow \) 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
Conclusion

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