Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols

François Dupressoir (Open University)
Andrew D. Gordon (Microsoft Research and University of Edinburgh)
Jan Jürjens (TU Dortmund)
David A. Naumann (Stevens Institute of Technology)

CSF 2011 – June 27
Abbaye des Vaux de Cernay

Contributions

- We describe a framework to:
  - prove C implementations of security protocols secure
  - in a symbolic model of cryptography
  - using a general-purpose verification tool
- We use our framework to verify simple examples:
  - a shared-key authenticated RPC protocol,
  - a variant of the Otway-Rees key exchange protocol

A Bit of Motivation

- Verifying C protocol code
  - Implementation flaws are detected at the same time as logical flaws
  - Security goals expressed as invariants and assertions in the code
- Using a general-purpose tool
  - In theory, developers may already be using them
  - Leverage recent advances in software verification
  - But, there are also drawbacks:
    - loss of automation and completeness,
    - informal statement of the verification result

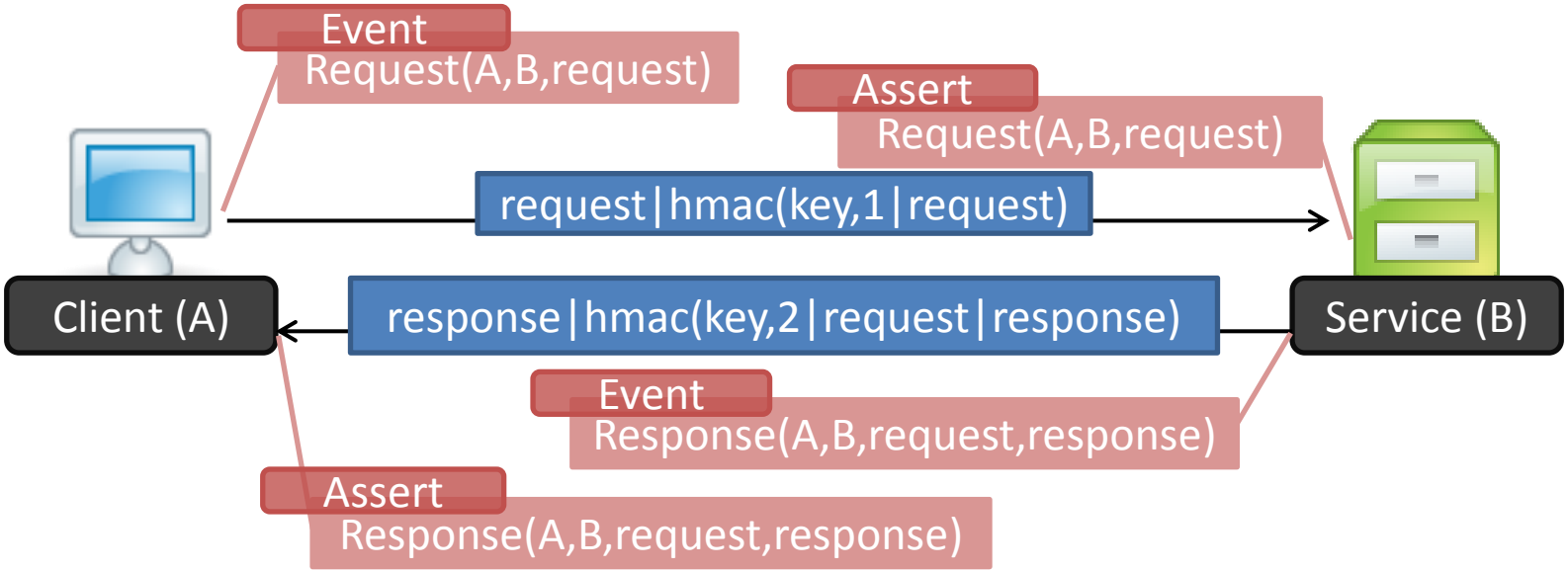
Prior Work

- Verifying C implementations:
  - Csur: Goubault-Larrecq & Parennes, 2005
  - ASPIER: Chaki & Datta, 2008
  - Pistachio: Udrea *et al.*, 2006 – compliance
  - Aizatulin *et al.*, Corin and Manzano – model extraction
- Verifying non-C implementations:
  - Java:
    - Jürjens, 2006
    - Elyjah/Hajyle (O'Shea, 2008)
    - Pironti & Sisto, Pironti & Jürjens – code generation
  - F#:
    - FS2PV/FS2CV (Bhargavan *et al.*, 2006)
    - F7 (Bhargavan *et al.*, 2007, 2010)

Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols

GOALS AND CHALLENGES

An Example: Authenticated RPC



A Problem with C

- The sha1_hmac() function in C takes two byte arrays and returns a fixed-length byte array
- Symbolic model:
  - hmac() is injective
  - byte arrays cannot be mapped back onto terms
- Computational model:
  - Properties are probabilistic
  - C needs to be given probabilistic semantics

Dealing with Collisions

- In this talk, we prove symbolic properties
- We will keep a one-to-one mapping between byte arrays and symbolic terms
- We enforce our symbolic assumptions when a byte array corresponds to two distinct terms

What We Want

- Given a verified implementation of a protocol role, instrumented with events and assertions

```
void RPC_client(bytes_c* alice, bytes_c* bob, bytes_c* key,
                bytes_c* req)
{  bytes_c *msg1, *msg2, *resp;

   Event(Request(alice,bob,req));
   msg1 = malloc(sizeof(*msg1)); if (msg1 == NULL) return;
   build_msg1(msg1,alice,bob,key,req);
   write(msg1);

   msg2 = malloc(sizeof(*msg2)); if (msg2 == NULL) return;
   read(msg2);
   resp = malloc(sizeof(*resp)); if (resp == NULL) return;
   if (parse_msg2(msg2,resp) == 0) return;
   Assert(Response(alice,bob,req,resp)); }
```

- And an arbitrary network attacker that controls the network and the scheduling, and can create new principals and run instances of the above mentioned implementation
- All the assertions hold at run time unless a collision has happened at a prior point in the execution

What VCC Proves

- Given a verified implementation of a protocol role, instrumented with events and assertions

```
void RPC_client(bytes_c* alice, bytes_c* bob, bytes_c* key,
                bytes_c* req)
{  bytes_c *msg1, *msg2, *resp;

   Event(Request(alice,bob,req));
   msg1 = malloc(sizeof(*msg1)); if (msg1 == NULL) return;
   build_msg1(msg1,alice,bob,key,req);
   write(msg1);

   msg2 = malloc(sizeof(*msg2)); if (msg2 == NULL) return;
   read(msg2);
   resp = malloc(sizeof(*resp)); if (resp == NULL) return;
   if (parse_msg2(msg2,resp) == 0) return;
   Assert(Response(alice,bob,req,resp)); }
```

- All assertions hold at run time* unless an assumption has failed at some prior point in the execution

* when the program is run in an environment where shared data is treated according to its specs and functions are called according to their specs

Program vs. Security Verification

- To get what we want from what VCC proves, we need:
  - A way of keeping track of which terms correspond to the byte arrays the C code manipulates, and enforcing its one-to-one-ness assumption
  - A way to model standard symbolic attackers as C programs that respect the shared state's invariants and only call functions in states where their preconditions hold
  - A model of symbolic cryptography in (VC)C

MODELLING SYMBOLIC CRYPTOGRAPHY IN C

Symbolic Cryptography in Coq: terms, events and log

- A standard term algebra

term ::= **Literal**(bs)
| **Pair**(t_1,t_2)
| **Hmac**(k,m)

- An algebra of events

event ::= **New**(t,usage)
| **Bad**(p)
| **Request**(a,b,req)
| **Response**(a,b,req,$resp$)

- Logs are sets of events, ordered with inclusion

Symbolic Cryptography in Coq: cryptographic invariants

- An inductive predicate MACSays()
  - Specifies what messages honest participants can MAC using a certain key

$$\frac{\mathbf{New}\big(k, \mathbf{KeyAB}(a,b)\big) \in \mathcal{L} \quad m = \mathbf{Pair}(\mathbf{Literal}(1), req) \quad \mathbf{Request}(a,b,req) \in \mathcal{L}}{\mathcal{L} \vdash \mathrm{MACSays}(k,m)}$$

- An inductive predicate Pub()
  - Specifies what messages may circulate in the clear over the network

$$\frac{\mathcal{L} \vdash \mathrm{MACSays}(k,m) \quad \mathcal{L} \vdash \mathrm{Pub}(m)}{\mathcal{L} \vdash \mathrm{Pub}\big(\mathbf{Hmac}(k,m)\big)} \qquad \frac{\mathcal{L} \vdash \mathrm{Pub}(k) \quad \mathcal{L} \vdash \mathrm{Pub}(m)}{\mathcal{L} \vdash \mathrm{Pub}\big(\mathbf{Hmac}(k,m)\big)}$$

- Some theorems to be used by the C verifier
  - Mostly inversion theorems, seen as *secrecy invariants*
  - But also monotonicity theorems

Symbolic Cryptography in C: terms, events and log

- Constructors become pure function symbols over aliases of the “mathint” type
- The log is a shared global mutable structure
  - models a set using boolean maps
  - initially empty
  - invariants (monotonic growth and some validity conditions) maintained throughout the execution
- Inductive predicates become pure functions
  - framed to depend monotonically on the log
  - axioms used to define them
- Theorems are imported as axioms

Mapping byte strings back to terms

THE TERM-BYTES TABLE AND HYBRID WRAPPERS

Term-Bytes Table

- Instrument the cryptographic functions to
  - dynamically maintain a one-to-one mapping
  - assume the symbolic assumptions are not violated
- Mapping seen as a dynamically growing table
  - represented as two inverse partial maps
  - shared global object
  - two-state invariants for growth
  - symbolic properties are invariants

The Hybrid Wrappers

- Put and verify symbolic cryptographic contracts on concrete cryptographic functions
  1. Perform the concrete operation
  2. Perform the symbolic computation
  3. Check for symbolic assumption violation
  4. Place the resulting term-bytes pair in the table
  5. Enforce the symbolic assumption (collision-freeness)

```
int hmacsha1(bytes *k, bytes *b, bytes *res
                    claimp(c))
    // Memory safety + table and log contracts
requires(MACSays(toTerm(k),toTerm(b))
        || (Pub(toTerm(k)) && Pub(toTerm(b))))
ensures(!result ==>
  toTerm(res) == Hmac(toTerm(k),toTerm(b)))
{
  spec(term tb,tk,th; bool collision = false;)

  // … (Allocate res)
  sha1_hmac(k->ptr, k->len, b->ptr, b->len, res->ptr);

  atomic(c, &table) {
     tb = toTerm(b);
     tk = toTerm(k);
     th = Hmac(tk,tb);

     if (Collision(res,th))
       collision = true;
     else
     { setTable(res,th); }})

  assume(!collision);

  return 0; }
```

The Symbolic Assumptions Summarized

- The log says that each “fresh” literal is given a unique usage
- The table
  - represents a finite bijection between a set of terms and a set of byte arrays
  - such that each term is associated with the bytestring corresponding to its concrete value

SYMBOLIC ATTACKERS AS C PROGRAMS*

* that treat shared data according to its specs and calls functions according to their specs

The Shim

- The attacker is a C interface
- Allows attack programs to start parallel role instances…
- … control the network
- … and keeps keys out of reach until compromised

```
session = setup(“Alice”,“Bob”);
clientC = getChannel_client(s);
serverC = getChannel_server(s);
run_server(s);
run_client(s,”Request”);
req = read(clientC);
write(serverC,req);
resp = read(serverC);
write(clientC,resp);
kAB = compromise_client(s);
```

Attacker Model Summarized

- Attack programs do not violate invariants or preconditions
  - Ensured using simple syntactic restrictions
- By VCC verification of the role code, we prove that no assertion can fail unless an assumption has failed
- Attack programs can represent all symbolic attacks, given a set of attacker capabilities

Performance on Simple Protocols

File/Function	LoC	LoA	Time (mins)
symcrypt.h	-	50	< 1
table.h	-	50	< 1
RPCdefs.h	-	250	< 1
ORdefs.h	-	250	< 1
hybrids.h	150	300	< 5
destruct()	20	40	< 5
hmacsha1()	20	20	< 1
RPCprot.c	130	80	< 15
client()	40	20	< 5
server()	40	10	< 10
ORprot.c	300	100	≈ 100
initiator()	40	15	< 5
responder()	100	100	≈ 60
server()	40	15	≈ 30

Summary

- We describe a general framework to guide a general-purpose C verifier to prove security properties of protocol implementations
  - We provide a first-order axiomatisation of cryptography, proved consistent in Coq
  - We give symbolic contracts to concrete cryptographic primitives by wrapping them in a hybrid layer
  - We model symbolic network attackers as restricted C programs
- We prove symbolic security of some small implementations of some small protocols up to symbolic cryptographic failures

The Future

- Getting a computationally sound verification method
- Providing a language independent front-end to generate the crypto invariants (Coq model and implementation headers)
- Other verifiers and other ways of using them