
François Dupressoir (Open University)
Andrew D. Gordon (MSR Cambridge)
Jan Jürjens (TU Dortmund)
Verifying Security Code

• Assume a security API specification.
• Implementing this API concretely may introduce bugs:
  – Wrong interpretation by the programmer.
  – Bugs that are lower-level than the specification’s abstraction.
• Verifying implementations addresses that issue.
  – fs2pv (CSF’06), Elyjah/Hajyle (ARSPA-WITS’08), F7 (CSF’08, POPL’10)
• But this verification should be done on programming languages that are used to write security code.
• We will focus on security protocol code written in C.
Verifying C Protocol Code

• C is used for performance, and in embedded systems.
• Fewer efforts on analysing C code:
  – Csur (VMCAI’05): trusted annotations, secrecy properties,
  – Aspier (CSF’09): trusted semantic description of subroutines, bounded settings, scalability issues,
  – Pistachio (USENIX’06): conformance verification only.
• We aim at verifying large security-critical projects in C.
  – Without too much manual work.
  – Without trusting user provided annotations.
• Our initial goal is to prove Dolev-Yao security, but we intend to later apply computational soundness results.
Our Method

• Using a general purpose verifier (VCC):
  – Bigger subset of the C language
  – Annotations are not trusted
  – Modular approach to verification

• Defining a Dolev-Yao attacker in the context of C programs.
  – Encoding the protocol state in the program state.
  – Encoding the attacker’s capabilities as invariants.
Our running example:

Authenticated RPC

Client

---

Service

---

request | hmac(key, request)

response | hmac(key, request | response)

Request(request)

Event

Response(request, response)

Assert

Response(request, response)
AN ATTACKER MODEL FOR C PROGRAMS
Attacker Model for C Programs

• We want to define a Dolev-Yao attacker model on C programs.
• F7 (POPL’10) defines a Dolev-Yao attacker model and a notion of security for the F# dialect of ML.
• Using the existing framework seems easier than defining an attacker directly on C.
• We could also benefit from future extensions to computational security.
Attacker Model for F7 Programs

- A type-checked F7 program forms a refined module, with imported ($I_I$) and exported ($I_E$) interfaces
- Result (POPL’10): In a refined module, if $I_I = \emptyset$, there is no type-safe way to use interface $I_E$ that makes any of the module’s assertions fail
The VCC Assumption
If a C program $P$ implements a header $h \llbracket I_E \rrbracket$ using a library $h \llbracket I_I \rrbracket$, then there exists an F7 module $\tilde{P}$ such that $I_I \vdash \tilde{P} \leadsto I_E$ (hence, no type-safe program using $I_E$ can make any of the assertions in $\tilde{P}$ fail).
Attacker Model For C Programs

• If a C program is verified to implement an exported header using an imported header, we assume an equivalent F7 refined module.
• An attacker is a well-typed F7 program that uses the exported interface.
• We use results from F7 (composition, security of refined modules...) to prove security of the verified C program.
Case Study:

A VERIFIED C IMPLEMENTATION OF AUTHENTICATED RPC
Authenticated RPC
A Reminder

- Client
  - Request(request)
  - Event Request(request)
- Service
  - Request(request)
  - Event Request(request)
  - Assert Response(request, response)
  - Assert Response(request, response)
- Request(request)
  - hmac(key, request)
- Response(request, response)
  - hmac(key, request | response)
  - Event Response(request, response)
Imported Libraries

- Primitives for network, byte array and cryptographic operations.
- Event predicate declarations.
- Inductive predicate definitions.
- For simplicity, we also include some protocol-specific functions.
Imported Interface

F7 function declaration

val hmacsha1:
  k:bytes →
  b:bytes {Bytes(b)
    ∧ ((MKey(k) ∧ MACSays(k,b))
    ∨ (Pub(k) ∧ Pub(b))}) →
  h:bytes {Bytes(h)
    ∧ IsMAC(h,k,b)}

VCC function contract

term hmacsha1_RCF(term k, term b)
  ensures(result != 0)
  requires(Bytes(b))
  requires((MKey(k) && MACSays(k,b))
    || (Pub(k) && Pub(b)))
  ensures(Bytes(result) && IsMAC(result,k,b));
Exported Interface

• The imported libraries

• The client role

  val client:
  a:bytes {String(a) ∧ Pub(a)} →
  b:bytes {String(b) ∧ Pub(b)} →
  k:bytes {Mkey(k) ∧ KeyAB(a,b,k)} →
  s:bytes {String(s) ∧ Pub(s)} →
  unit

  void client(term a, term b, term k, term s)
  requires(String(a) && Pub(a))
  requires(String(b) && Pub(b))
  requires(Mkey(k) && log->KeyAB[k][a][b])
  requires(String(s) && Pub(s))
  ensures(Stable(log));

• The server role

  val server:
  a:bytes {String(a) ∧ Pub(a)} →
  b:bytes {String(b) ∧ Pub(b)} →
  k:bytes {Mkey(k) ∧ KeyAB(a,b,k)} →
  unit

  void server(term a, term b, term k)
  requires(String(a) && Pub(a))
  requires(String(b) && Pub(b))
  requires(Mkey(k) && log->KeyAB[k][a][b])
  ensures(Stable(log));
Implementation

• To simplify memory-safety, we wrap byte arrays.

```c
struct {
    unsigned char *ptr;
    unsigned long len;
} bytes;
```

• We use ghost fields to specify their usage.

```c
struct {
    unsigned char *ptr;
    unsigned long len;

    spec(mathint encoding);
    invariant(keeps(as_array(ptr,len)))
    invariant(encoding == Encode(ptr,len))
} bytes;
```

• Encode() is a bijective encoding of byte arrays as integers, so we can talk about byte arrays as values.
Implementation

```c
void client(bytes_c *a, bytes_c *b, bytes_c *k, bytes_c *s)
{
    bytes_c *req,*mac,*upay,*msg1;
    bytes_c *msg2,pload2,mac2,*t,*resp;
    int res;

    if ((req = malloc(sizeof(*req))) == NULL)
        return;
    if (request(s, req))
        return;

    if ((mac = malloc(sizeof(*mac))) == NULL)
        return;
    if (hmacsha1(k, req, mac))
        return;

    free(req);

    if ((upay = malloc(sizeof(*upay))) == NULL)
        return;
    if (utf8(s, upay))
        return;

    if ((msg1 = malloc(sizeof(*msg1))) == NULL)
        return;
    if (concat(upay, mac, msg1))
        return;

    free(upay);
    free(mac);

    send(msg1);
    free(msg1);
}
```

```c
if ((msg2 = malloc(sizeof(*msg2))) == NULL)
    return;
if (recv(msg2))
    return;

if (iconcat(msg2, &pload2, &mac2))
    free(msg2);

if ((t = malloc(sizeof(*t))) == NULL)
    return;
if (iutf8(&pload2, t))
    return;

if ((resp = malloc(sizeof(*resp))) == NULL)
    return;
if (response(s, t, resp))
    return;

free(t);
free(s);

if (!hmacsha1Verify(k, resp, &mac2))
    return;
free(resp);
```
Hybrid Wrappers

- Two goals:
  - Provide a concrete interface for realistic C code
  - Ensure consistency between the VCC axioms and our cryptographic definitions

- They are wrappers around both the concrete functions (e.g. OpenSSL crypto library) and the symbolic functions imported from RCF.

- Wrappers are verified so that:
  - The concrete part does not introduce run-time errors
  - The symbolic part follows the cryptographic invariants
Hybrid Representation

- The encoding function is a mapping from arrays of bytes to mathematical integers
- The F7 functions manipulate Dolev-Yao terms
- We keep the models in lock-step using two partial maps:
  
  
  term B2T[bytes];
  bytes T2B[term];

  invariant(forall(bytes b; B2T[b] != 0 ==> T2B[B2T[b]] == b)
  invariant(forall(term t; T2B[t] != 0 ==> B2T[T2B[t]] == t)
Constructing a refined module

- What we have:
  - A refined module $(\emptyset, \text{Lib}, \text{Lib})$, where $\text{Lib}$ contains network and cryptographic functions, the predicate definitions, and the request, response, and service functions (POPL’10).
  - Via the VCC assumption, a refined module $(\text{Lib}, P, \text{RPC}^-)$, where $\text{RPC}^- = \text{Lib}$; \text{val} client: ...; \text{val} server:...
Constructing a refined module

• We can write, in F7, a wrapper function:

```ocaml
let setup (a:bytes {String(a)}) (b:bytes {String(b)}) =
  let k = mkKeyAB a b in
  (fun s -> client a b k s),
  (fun _ -> server a b k),
  (fun _ -> assume(Bad(a)); k),
  (fun _ -> assume(Bad(b)); k)
```

• And by composition, we can build a refined module \((\emptyset, M, (I_L, I_R))\), where \((I_L, I_R)\) is the attacker interface defined in (POPL’10), and provides the attacker with control over the network and the principals (through the setup function).
Summary

• We define a Dolev-Yao attacker model for C programs and define the corresponding notion of security.

• We verify that a program is secure under certain assumptions:
  – The VCC assumption.
  – The assumption that cryptographic primitives fail instead of generating colliding byte arrays.
Summary – Case Study

• The RPC protocol roles (~60 LoC) are verified in about 3 minutes each.

• Necessary annotations:
  – Pre-conditions on the protocol roles (memory-safety + translated from F7): ~10 lines per role.
  – Library contracts (memory-safety + translated from F7): ~10 lines per function prototype.
  – Hybrid wrappers (20-50 lines per primitive depending on the library).
  – Some hints to the prover: < 10 lines per role, depending on memory behaviour.

• A lot of the annotations are memory-safety related.
• Some hybrid wrappers can be a lot of trouble (concatenation).
Future Work

• Weaken our assumptions.
• Compare this approach with more direct encodings of the attacker in VCC:
  – Performance (e.g. no inductive predicates).
  – Simplicity of the security result.
• Adapt to a computationally sound model:
  – Eliminate the hybrid wrappers and functional idioms.
  – Get computational guarantees.
• Apply to externally written code:
  – Protocols (e.g. PolarSSL).
  – Security API implementations? Suggestions are welcome.