Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols

François Dupressoir, Andrew D. Gordon,
Jan Jürjens, David A. Naumann

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
- Our methods include:
  - describing cryptography using inductive predicates
  - encoding the protocol state as shared program state
  - expressing security goals as assertions in the code
  - expressing security assumptions as assumptions in the code
  - describing symbolic attacks as restricted C programs
- We use our framework to verify simple examples:
  - a shared-key authenticated RPC protocol,
  - a variant of the Otway-Rees key exchange protocol

WHY?

A Brief History of Security Flaws

- Security protocols are difficult to get right
- Formal tools developed to automate reasoning about them:
  - Dolev & Yao, 1982: foundations of symbolic crypto
  - Lowe, 1995: used CSP to prove security protocols symbolically secure
  - Blanchet, 2001: ProVerif
  - More recently: CryptoVerif, CertiCrypt
- But application seems problematic:
  - Kerberos, TLS were deployed despite symbolic flaws
  - Existing tools analyse models, not code

Focusing on SSL and openSSL

- 1996: SSLv3 is released, fixing security flaws in SSLv2, including symbolic flaws
- December 1998: openSSL is first released
- December 2008: major implementation flaw (dating back to 1998)

```
i = EVP_VerifyFinal(&md_ctx,p,(int)n,pkey);
if (i)
{ /* do good things */ }
```

  but EVP_VerifyFinal returns -1 when the certificate is malformed…
- December 2009: symbolic attack on SSL is found

Verifying C Protocol Code

- No need for a separate abstract model of the protocol
  - The C code is the protocol description
  - Security goals are expressed as invariants and assertions in the code itself
- Implementation flaws are detected at the same time as logical flaws
- What we are not planning on preventing:
  - Non-symbolic attacks: timing, power consumption…
  - Non-network attacks: buffer overflows, physical attacks…

Prior Work

- Verifying non-C implementations:
  - F#:
    - FS2PV/FS2CV (Bhargavan *et al.*, 2006)
    - F7 (Bhargavan *et al.*, 2007, 2010)
  - Java:
    - Jürjens, 2006
    - Elyjah/Hajyle (O’Shea, 2008)
    - Pironti *et al.* – automatic generation of monitor code
- Verifying C implementations:
  - Csur: Goubault-Larrecq & Parennes, 2005
  - ASPIER: Chaki & Datta, 2008
  - Pistachio: Udrea *et al.*, 2006 - compliance

Using a General-Purpose Tool

+ No need for the programmer to learn a new language (ideally…)
+ They exist, and are advancing and multiplying rapidly (Caduceus/Frama-C, VCC , VeriFast…)
• No extra assumptions about the program (memory-safety…)
- Informal description of what is proved
- Greater loss of automation and completeness

A Quick Word On

GENERAL-PURPOSE VERIFICATION

Hoare Logic and Verification-Condition Generation

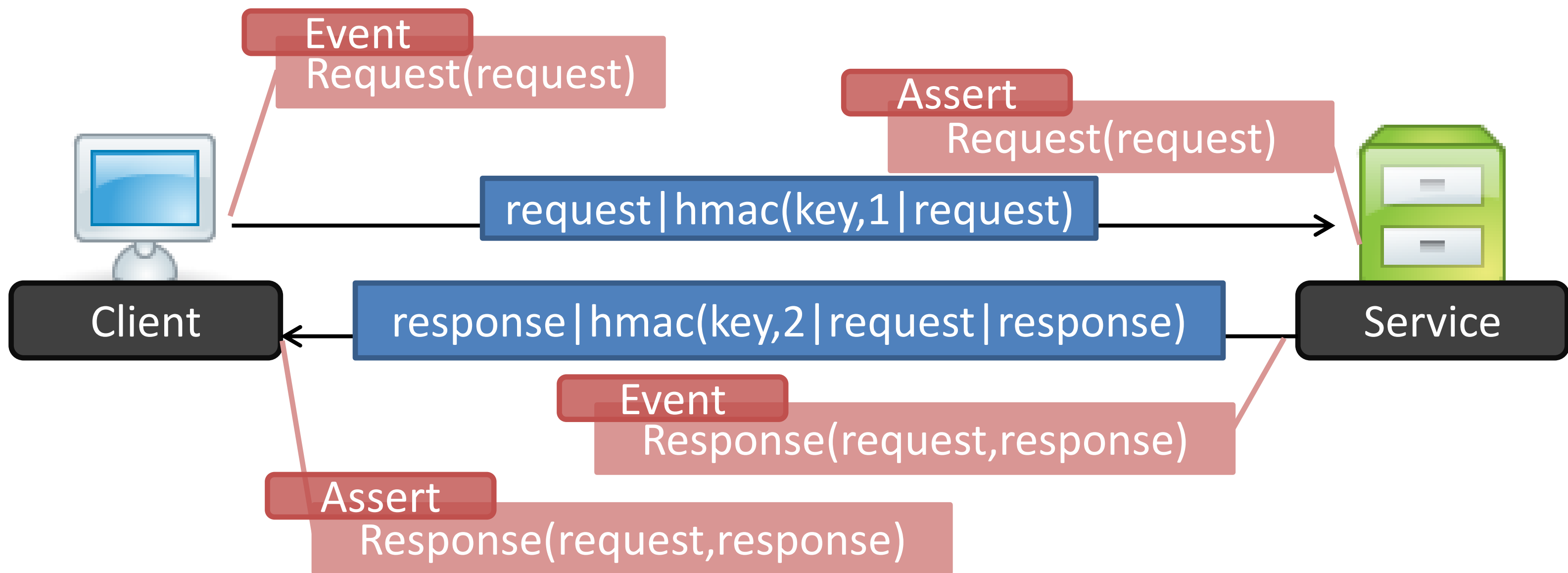
- Each command is given a pre-condition and a post-condition
  - When called in a state in which the pre-condition holds…
  - … the command returns a state in which the post-condition holds
- At desired program points, the user inserts assertions about the state (e.g. the value stored in x is greater than 2)
- The program is traversed backwards to find the weakest pre-condition that ensures the assertion holds
- If the weakest pre-condition is always true, then the assertion will always hold

VCC and VC Generation

- VCC deals with C programs (function calls, memory heap, addressable local variables…)
- VCC deals with state-sharing concurrent C programs
  - Typed memory model preventing partial overlaps
  - Objects are equipped with invariants expressing how concurrent threads can modify them
  - Memory reads and updates are guarded by accessibility checks
- But this all works along the lines of the general idea

WHAT WE PROVE

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
- The symbolic security property for MACs says that the hmac() function is injective
  - C byte arrays cannot be naively mapped back to symbolic terms
- The computational security property for MACs says that the probability of any probabilistic polynomial time Turing machine being able to forge a valid MAC is negligible
  - C needs to be given probabilistic semantics
  - The verification tool needs to understand it

Dealing with Collisions

- Two possibilities:
  - Express security properties directly on byte arrays
    - Probabilistic properties → need probabilistic semantics and tools for C
  - Keep a one-to-one mapping between byte arrays and symbolic terms
    - Enforce symbolic assumptions when a byte array corresponds to two distinct terms
- We choose the second option.

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 enforce 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

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

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

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

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

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

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
  - one boolean map per event predicate
  - 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

The 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 concretely as two maps
  - shared global object
  - two-state invariants for growth
  - symbolic properties are invariants

The Hybrid Wrappers

- Ideally, have the same prototype as the concrete functions
  - We provide a wrapped type for byte arrays
- Put and verify symbolic cryptographic contracts on concrete cryptographic functions
  1. Perform the concrete operation
  2. Look up the input terms in the table
  3. Check for symbolic assumption violation
  4. Place the resulting term-bytes pair in the table

Symbolic Assumptions

- 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 bytearrays
  - 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

Attacker Capabilities

- We want our attackers to:
  - Control the network
  - Create new principals
  - Instantiate principals with protocol roles
  - Compromise principals
  - Control the scheduling
- We do not want our attackers to:
  - Look into protocol role memory
- We want the VCC verification result to mean something about security

The Shim

- The attacker is a C interface
  - <u>first-order</u>
  - C has virtually no type abstraction
  - VCC has literally no type abstraction
- Interface needs to do the forking…
  - And all the ownership transfer and memory safety checks that go with it
- … and keep keys out of reach until compromised
  - Protocol-specific part is session-based
- Simple syntactic restrictions ensure attack programs do not violate invariants or preconditions

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
- Long version: http://research.microsoft.com/en-us/um/people/adg/guiding-full.pdf
- RPC code: http://research.microsoft.com/~adg/guiding-source.tar.gz

The Future

- Currently looking at PolarSSL, a very small implementation of SSL/TLS
  - invariants have been developed for the record layer by Bhargavan and Pironti
- Providing a language-independent front-end to generate the crypto invariants from a type-based description of the protocol messages
- Other verifiers (different concurrency models), other ways of using them, “simply” verifying compliance
- Getting closer to computational soundness by building on top of Fournet’s computationally sound F7 interface