Proving Security Properties of C Programs Using VCC

François Dupressoir
FMATS workshop
7-8 December 2011, Cambridge

Problem

- Given a C program that uses cryptography, prove that it enjoys certain security properties (authentication, confidentiality)
- We will assume security and correctness of the cryptographic primitive implementations
- Start with symbolic models of cryptography, and generalize to computational models

Related Work

- Model Extraction: whole-program analysis, no specification needed
  - Csur (Goubault-Larrecq & Parrennes, 2005)
  - Aspier (Chaki & Datta, 2009)
  - Csec-Modex (Aizatulin et al., 2011)
  - Elyjah (O'Shea), FS2PV/CV (Bhargavan et al.)
- Security by Typing: local, invariant-based analysis, specification needed
  - F7 (Bhargavan et al., 2008)
    - Invariants on global log encode acceptable use of cryptography
    - Refinement types used to verify the program respects the invariants
  - Invariants can model symbolic crypto or ideal functionalities (Fournet et al., 2011)

General-Purpose C Verification

- Advantages:
  - Benefit from the properties of existing tools:
    - parsing, semantic peculiarities…
    - modularity
    - soundness for trace properties
  - Benefit from future tool developments:
    - performance improvements
    - new features (relational properties, information-flow…)
- Drawbacks:
  - Everything is proved by the tool (annotation cost)
  - Legacy code may be difficult to deal with (understand)

CSF 2011, with A. Gordon, J. Jürjens and D. Naumann

SYMBOLIC SECURITY

Motivation

- Have automated tool support:
  - TAPS, ProVerif, LySa…
- Can be introduced by developers:
  - OpenSSL signature API misuse (January 2009)
- Despite tool support, bugs still appear in recent protocol specifications (HTTPS, TPM)

Some Notes

- We prove authentication as non-injective correspondences
  - If event End happens, then event Begin has happened in the past
- We do not use end events, we assert where desired that a begin event has been executed
- We prove weak secrecy (full disclosure)

Cryptographic Model

- Build an inductive model of the cryptography used in the protocol:
  - Literals are given a unique usage
  - Protocol events and key creation and compromise are logged in a set of events
  - Usages and log yield several predicates
    - “Payload k p Log”: k can be used to protect p in state Log
    - “Release k Log”: k can be released to attacker in state Log
  - Payload predicate will serve as precondition to cryptographic operations
- Security properties are theorems in this model
  - We can build and prove in Coq
  - Or we can try to do it in VCC

Proving Security of C Programs

- Security model and theorems as ghost code
- Formalizing symbolic assumptions:
  - Symbolic cryptography works on algebraic terms
  - C code manipulates bounded bytestrings
  - We keep a ghost table ensuring a one-to-one mapping
- Modelling the attacker:
  - All symbolic attackers should be considered
  - VCC only guarantees soundness when the whole process/environment is verified
  - We convince you that any symbolic attacker can be written as a C program and verified
- By refinement (Polikarpova & Moskal, VSTTE 2012)

Example Security Result

Attacker Shim

```
typedef bytespub;

bytespub* att_toBytespub(unsigned char* ptr,
                         unsigned long len);
bytespub* att_pair(bytespub* b1, bytespub* b2);
bytespub* att_fst(bytespub* b);
bytespub* att_snd(bytespub* b);

bytespub* att_hmacsha1(bytespub* k, bytespub* b);
bool att_hmacsha1Verify(bytespub* k,
                        bytespub* b,
                        bytespub* m);

void att_channel_write(channel* chan, bytespub* b);
bytespub* att_channel_read(channel* chan);

typedef session;

session* att_setup(bytespub* cl, bytespub* se);

void att_run_client(session* s, bytespub* request);
void att_run_server(session* s);

bytespub* att_compromise_client(session*s);
bytespub* att_compromise_server(session*s);

channel* att_getChannel_client(session* s);
channel* att_getChannel_server(session* s);
```

Partial Client Code

```
void client(bytes_c *alice, bytes_c *bob, bytes_c *kab, bytes_c *req, channel* chan)
{
  bytes_c *toMAC1, *mac1, *msg1;
  bytes_c *msg2, *resp, *toMAC2, *mac2;
  Event(tmp,log.Request[table.B2T[alice->encoding]]
                       [table.B2T[bob->encoding]]
                       [table.B2T[req->encoding]]);

  /* ... Build and send request ... */

  if ((msg2 = malloc(sizeof(*msg2))) == NULL)
    return;
  if (channel_read(chan, msg2 spec(freshClaim(c))))
    return;

  if ((resp = malloc(sizeof(*resp))) == NULL)
    return;
  if ((mac2 = malloc(sizeof(*mac2))) == NULL)
    return;
  if (destruct(msg2, resp, mac2 spec(freshClaim(c))))
    return;

  if ((toMAC2 = malloc(sizeof(*toMAC2))) == NULL)
    return;
  if (response(req, resp, toMAC2 spec(freshClaim(c))))
    return;

  if (!hmacsha1Verify(kab, toMAC2, mac2 spec(freshClaim(c))))
    return;
  assert(log.Response[table.B2T[alice->encoding]]
                     [table.B2T[bob->encoding]]
                     [table.B2T[req->encoding]]
                     [table.B2T[resp->encoding]] ||
         log.Bad[table.B2T[alice->encoding]] ||
         log.Bad[table.B2T[bob->encoding]]);
}
```

Experimental Results

- In Dupressoir et al, CSF'11:
  - HMAC-based authenticated RPC:
    - ~150 LoC, ~1 LoA/LoC + model, < 10 minutes
  - Otway-Rees:
    - ~300 LoC, ~1 LoA/LoC + model, ~1 hour
- In Aizatulin et al, FAST'11:
  - Encryption-based authenticated RPC:
    - Written to be challenging (parsing is inlined, crypto is hard)
    - ~300 LoC, ~1 LoA/LoC + model
      - most functions: < 10 s
      - request sending: ~10 min
      - request parsing: runs out of memory

Remaining Problems

- Verification: performance is an issue
  - we could specialize contracts, but lose modularity
- It is relatively easy for the attacker to violate the symbolic assumptions
  - for example, concrete format of pairs is known
- Weak secrecy may not be the most realistic notion of secrecy for protocols
  - partial leakage is not considered
  - a more realistic notion: indistinguishability (observational equivalence)

Work in progress,

with Ernie Cohen, Cédric Fournet, Andy Gordon and Michał Moskal

COMPUTATIONAL SECURITY

Computational Cryptography

- Adversary: polynomial-time probabilistic program
- Security properties are negative and probabilistic:
  - An adversary that has access to a signing oracle can only forge a new signature for a message with negligible probability (INT-CMA)
  - An adversary that has access to a left-right encryption oracle can only distinguish between the left and right implementations with negligible probability (IND-CPA)
- We need to remove all concurrency: network send and receive become control primitives

Ideal Functionalities

- Given concrete cryptographic functions, build an idealized version that is trivially secure
  - For example, ideal encryption encrypts zeroes instead of the plaintext (and decryption is a table lookup)
- The assumption is that the ideal functionality cannot be distinguished from the concrete one
- Ideal functionalities can often be given types suitable for security verification by typing
- It works in F# (Fournet et al., 2011)

But… Why?

- To get stronger security guarantees, we need to look at indistinguishability properties
- Given ideal functionalities, the real difficulty is in proving that the only flows from the secrets to the adversary are through the cryptography
  - this is indistinguishability
  - note that this is not “absence of flows”
- If we do things properly, we still get to fall back on symbolic cryptography if we fail

Proving Indistinguishability on C Programs

- Ideally, relational verification:
  - Assertions, post-conditions over pairs of runs
  - No tool support, benefit/cost rather low for general-purpose verifiers to implement
- The work on F# uses type parametricity:
  - The return value of a function cannot depend on the value of an argument that is abstractly typed
  - Types in C?
  - What happens when the memory doesn’t get wiped?
  - There are more fun things going on

A Solution

- Write an abstract version of the code
  - Ghost VCC code
  - Operates on values, not memory
  - Has abstract types (perhaps even parametricity?)
- Somehow verify indistinguishability properties on the abstract code
  - Thinking of translating between VCC and F#
- Prove that the C code is a precise refinement of its ghost abstraction
  - $f_c(in) = \gamma\left(f_a\left(\alpha(in)\right)\right)$
  - Need to capture all adversary channels (network, errors…)

A (non-ideal) Solution

- All functions in the system under study need to be deterministic
  - Probabilities don’t count
  - But implementation-specific stuff gets in the spec
- All functions in the system need to be looked at in that much detail
  - Even the one that reports the chip’s capabilities?
- Reviving the flow analysis would help

A (pretty good) Solution: A Scenario

- You are developing a specification for a new security-critical piece of software/hardware
- Someone has been pushing for
  - An executable spec to reduce ambiguities
  - Some formal guarantees
- Write
  - A formal spec in F#, along with a computational security proof, intended to convince academics
  - A C implementation, verified to precisely refine the F# specification, intended to be used by developers

Conclusion

- We can prove symbolic security properties of C code that uses cryptography
  - Room for performance improvements
  - Could use automated inference of memory-safety
  - This is not VCC specific
- We are making progress towards proofs of computational security properties
  - Main problem lies in proving non-trace properties using a trace property verifier
- We are looking for small pieces of real code
  - http://research.microsoft.com/en-us/projects/csec-challenge/