Formal Verification of Cryptographic Software Implementations

Bárbara Vieira [1], J. Bacelar Almeida [1]
Manuel Barbosa [1] and Jorge S. Pinto [1]

[1] Universidade do Minho
Departmento de Informática
HASLab / INESC TEC

December, 2011

Why should we formally verify cryptographic software?

- Cryptographic algorithms are usually given as high-level specifications;
- Specifications are based on mathematical constructions which do not directly map into programming language structures (mathematical fields, arbitrary precision integers, etc);
- Implementations of cryptographic algorithms are by themselves complicated;
- To obtain high performance in different platforms, cryptographic algorithms are optimised;
- Optimizations can introduce errors and compromise the security of the algorithms.

How can we formally verify cryptographic software?

1. Establishing the **security properties** that a cryptographic software implementation must enforce; *Eg. memory safety, data confidentiality, etc.*

2. Applying **formal techniques** that can be used to verify if the software implementations indeed enforce the desired security properties;

Security properties

So far, we have addressed:

- Safety properties memory safety (e.g. absence of buffer overflows); arithmetic safety (e.g. absence of integer overflows);
- Error propagation Analysing the behavior of stream ciphers when a bit in the ciphertext is flipped over the communication channel;
- Functional correctness Verifying the correctness of cryptographic algorithm's implementations with respect to a reference implementation (the specification acts as a reference implementation – *code refactoring*);
- Minimising exposure to side-channel attacks Verifying if the implementations of cryptographic algorithms satisfy security properties which minimise exposure against certain side-channel attacks.

Deductive verification

- Formal verification technique which relies on *Hoare Logic*;
- Aims to establish correctness in software systems;
- It is based on the *Design by Contract* approach (pre- and post-conditions);

Our motivation on the use of deductive verification

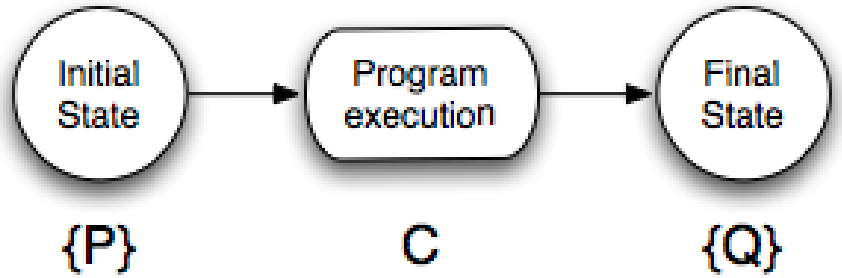
- One unified methodology to deal with a wide range of security properties;
- It can be used to verify security relevant properties using well-known verifications tools;
- It demonstrates a great potential to verify noninterference-like properties using the self-composition approach;

Verification platforms

Verification platforms based on *Hoare logic*

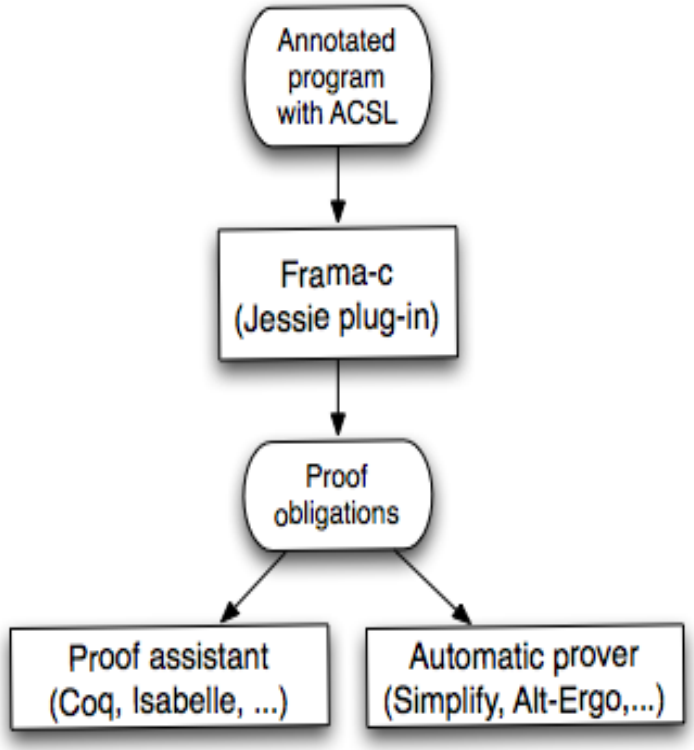
- Annotation language: allows reasoning about program executions – specifications are introduced using *Hoare triples*:$\{P\}\ C\ \{Q\}$
- Verification condition generator (VCGen): from an annotated program, it generates a set of proof obligations
- Proof obligation: formulas in first-order logic whose validity implies that the software meets its specification

Hoare triple specification



Frama-C

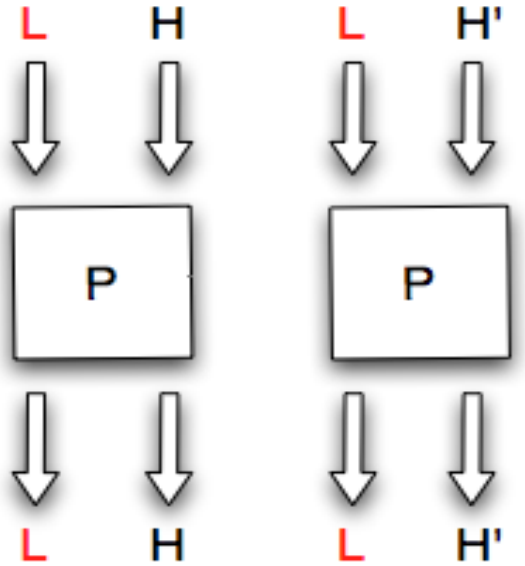
- Framework for static analysis of C programs;
- Includes the `Jessie` plug-in to make deductive verification;
- The specification language – ACSL (mostly inspired by JML);
- Automatically generates proof-obligations associated with memory safety and absence of integer overflows.



Noninterference

Informal definition

A program satisfies noninterference if high inputs do not interference with the computation of low outputs.



Self-composition

Barthe et al. observed that noninterference of a program P can be reduced to a property about a single program execution of the program $P; P'$, where P' is the re-named copy of P.

$$\{L = L'\}\ P; P'\ \{L = L'\}$$

NaCl security policies

NaCl[1] cryptographic library countermeasures

NaCl developers observed that to minimise exposure to side-channel attacks it suffices that cryptographic implementations satisfy:

- No data-dependent branches – there are no conditional branches and loops with conditions based on input data;
- No data-dependent array indices – there are no array lookups with indices based on input data;

Goal

Formally verify if the NaCl cryptographic library attests adherence to these side-channel countermeasures.

Adopted strategy

Formalise these policies as noninterference properties.

[1] http://nacl.cr.yp.to

Minimising exposure to side-channel attacks

Side-channel attack

Any attack that takes advantage of observing specific characteristics of the physical implementations of cryptographic algorithms;

Examples of addressed timing side-channel attacks

- Cache timing attacks – attacks exploiting the time that a computation (in the cache) takes to perform;
- Branch prediction analysis attacks – attacks exploiting secret information that can be leaked through conditional branches;

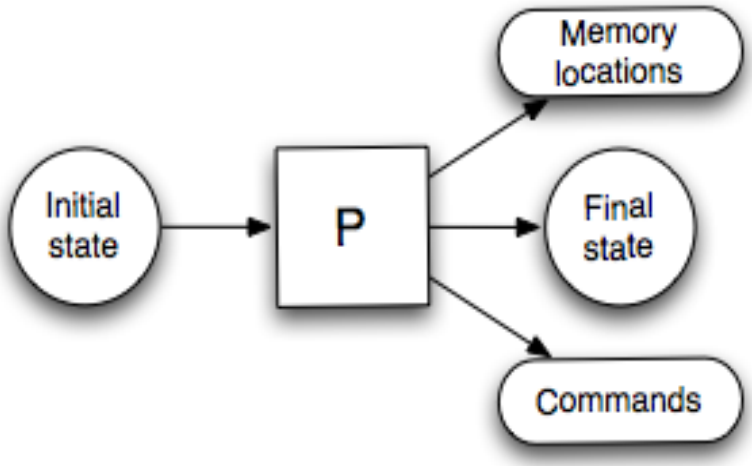
Informal security property

An attacker capable of observing the instruction pointer and accessed memory locations cannot recover any sensitive information.

Formalising side-channel countermeasures as noninterference (1)

Extending program semantics

Extending program semantics to capture in the post-state the accessed memory locations and the executed commands.



Formally

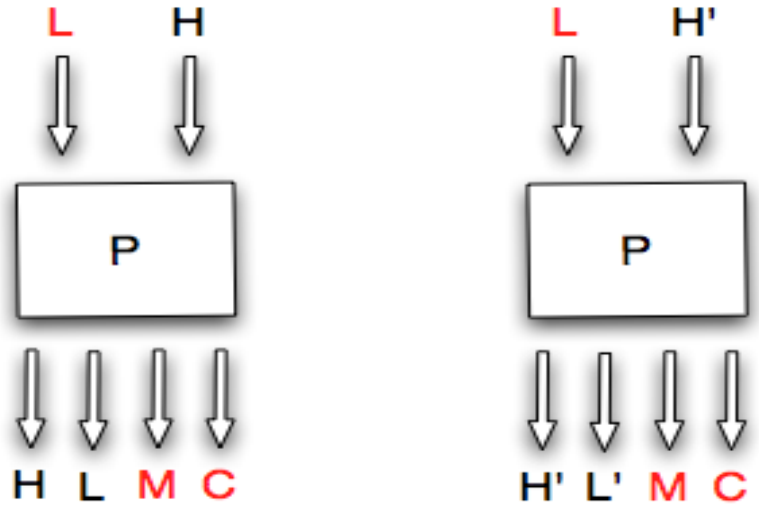
Extended program semantics – $(P, S) \Downarrow (S', M, C)$[2], where

- M - list of memory locations accessed during program execution;
- C - list of commands executed by the program during its execution;

[2] Standard program (big-step) semantics – $(P, S) \Downarrow S'$;

Formalising side-channel countermeasures as noninterference (2)

For low-equal initial states, executing two instances of the same program, they must produce the same accessed memory locations and execute the same sequence of commands.



Security property formalised as noninterference

Low integrity inputs should not interfere with the accessed memory locations neither with the executed commands.

Verifying side-channel countermeasures using self-composition

To express security directly over the program state we transform the original program to include sufficient trace information[3]:

- Control-flow – list containing the evaluation of the conditions of all conditional branches and loops;
- Memory access – for each array variable is created a list containing the accessed array indexes during program execution;

Security property formalisation (using Hoare triples)

$$\left\{L = L'\right\}\ P; P'\ \left\{\forall x.\ M_x = M'_x \wedge C = C'\right\}$$

where

- M_x, M'_x – lists containing the accessed indexes in array x
- C, C' – lists containing the evaluation of the conditions of all conditional branches and loops;

[3] Recall that memory and control-flow traces are not part of the program state;

Verifying side-channel countermeasures using Frama-c

- Transform the original program to include two different kind of lists as ghost variables: control-flow list and array access lists;
- If the code includes loops, each loop invariant must also refer the ghost variables;
- Annotate the program with pre- and post-conditions to express the security definition;
- Use `Frama-C` to automatically discharge all the proof obligations

Example

mulmod function (also extracted from the NaCl core library) computes the modular multiplication operation.

```
static void mulmod(unsigned int h[17],const unsigned int r[17]) {
  unsigned int hr[17]; unsigned int i; unsigned int j; unsigned int u;
  for (i = 0;i < 17;++i) {
    u = 0;
    for (j = 0;j <= i;++j) u += h[j] * r[i - j];
    for (j = i + 1;j < 17;++j) u += 320 * h[j] * r[i + 17 - j];
    hr[i] = u;
  }
  for (i = 0;i < 17;++i) h[i] = hr[i];
  squeeze(h);
}
```

Example (internalising trace information)

```
static void mulmod(unsigned int h[17], const unsigned int r[17]) {
    unsigned int hr[17]; unsigned int i; unsigned int j; unsigned int u;

    for (i = 0; i < 17; ++i) {
      u = 0;
      for (j = 0; j <= i; ++j) { u += h[j] * r[i    j];
          //@ ghost append_h(j);  append_r(i-j); append_cflow(j<=i); }
      //@ ghost append_cflow(j<=i);

       for (k = i + 1; k < 17; ++k) { u += 320 * h[k] * r[i + 17 - k];
           //@ ghost append_h(k); append_r(i+17-k); append_cflow(k<17); }
       //@ ghost append_cflow(k<17);
       hr[i] = u;
       //@ ghost append_hr(i);  append_cflow(i<17);}
       //@ ghost append_cflow(i<17);

   for (i = 0; i < 17; ++i) { h[i] = hr[i];
       //@ ghost append_h(i); append_hr(i); append_cflow(i<17);}
   //@ ghost append_cflow(i<17);
   squeeze(h);
   //@ ghost append_h(0);
}
```

Example (pre- and post-conditions)

```
/*@ requires lmem_h == lmem_h1;
  @ requires lmem_hr == lmem_hr1;
  @ requires lmem_r == lmem_r1;
  @ requires lmem_cflow == lmem_cflow1;
  @ ensures lmem_h ==lmem_h1 &&  lmem_r == lmem_r1 &&
  @                  lmem_hr ==lmem_hr1 && lmem_cflow == lmem_cflow1;
  @*/
```

Summary and conclusions

- Deductive verification techniques help to improve the development of cryptographic software, by reducing the error rating and giving better guarantees that the software indeed behaves as prescribed;
- We have demonstrated how the NaCl security policies can be formalised and verified using tools such as the `Jessie` plug-in from the `Frama-C` framework;
- Further directions: study how the annotations process can be automated (since it looks like it is simple and amenable of optimisation).

Bibliography


J. Bacelar Almeida , M. Barbosa, J. Sousa Pinto and Bárbara Vieira
Deductive verification of cryptographic software.
NASA Journal of Innovations in Systems and Software Engineering, 2010.


J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira
Formal verification of side-channel countermeasures using self-composition.
Science of Computer Programming, 2011.