Formal verification of security policies
of cryptographic software

Bárbara Vieira

HasLab/Departamento de Informática
Universidade do Minho
Campus de Gualtar,
Braga, Portugal

Maio, 2011

Joint work with:
Manuel Barbosa, Jorge S. Pinto,
J. Bacelar Almeida, J.C. Filliâtre
Formal verification key concepts

- **Formal methods** aim at the design of mathematically-based techniques for the specification, development and verification of software/hardware systems.
- **Formal verification** is the formal methods area whose purpose is establishing the correctness of software systems, by proving that the programs satisfy (or not) a certain pre-defined specification.

Why should we establish the correctness of software systems?

- Avoid errors – avoid losing human lives, losing money, etc;
- Have a mathematical proof that the software indeed works as prescribed.
Motivation

Ariane 5 explosion (1996)

Arithmetic overflow
Trying to represent 64-bit floating point in a 16-bit signed integer\(^1\).

\(^1\)http://en.wikipedia.org/wiki/Ariane_5_Flight_501
Console game exploits

Wii exploits (2008)

Motivation

Heap overflow
The overflow occurs when an application copies more data, than the buffer was designed to contain. The routine is not verifying that the source fits into the destination.

(\text{http://www.thexploit.com/secnews/ps3-heap-overflow-exploit-explained/})

Stack overflow
An untrusted program writes more data to a buffer located on the stack than there was actually allocated for that buffer. It corrupts the stack by injecting the malicious executable code into the running program.

(\text{http://en.wikipedia.org/wiki/Twilight\_hack})
Outline

Motivation

Formal verification and Cryptography
Deductive verification

Security policies verification using Frama-C
  Safety properties
  Error propagation
  Programs equivalence
  Adherence to side-channel countermeasures

CAOVerif: A Deductive Verification Tool for CAO
  CAO language
  Architecture

Conclusions and Future work
Part I

Key concepts
Why should we apply formal verification techniques to verify cryptographic software?

▶ Cryptographic algorithms are usually given as specifications;
▶ Specifications are based on mathematical constructions which do not directly map into programming language structures (mathematical fields, arbitrary precision integers, etc);
▶ Because of speed issues, optimizations are introduced in the implementations;
▶ Optimizations can introduce errors and compromise the security of the algorithms.

Remark: We are not particularly interested in Cryptanalysis!
Deductive verification

- Aims to establish correctness in software systems
- It is based on the *Design by Contract* approach (pre- and post-conditions)
- It relies on the use of logical formulas whose validity implies the correctness of the original program
- **Advantages:**
  - Helps finding errors in the program and correcting them
  - Generalises a verified algorithm to capture new cases that were not anticipated (e.g. finding pointer de-referencing)
  - Gives a better understanding of the verified algorithm
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 triples

\[
\begin{align*}
\text{Initial state} & \quad \rightarrow \quad \text{Execution} \quad \rightarrow \quad \text{Final state} \\
\{P\} & \quad \rightarrow \quad C & \quad \rightarrow \quad \{Q\}
\end{align*}
\]
Frama-C

- Framework for static analysis of C programs
- Includes a plug-in – JESSIE plug-in build on the top of Why platform – to make deductive verification
- The specification language – ACSL – is mostly inspired by JML
- Automatically generates proof-obligations associated with memory safety and absence of integer overflows

Methodology
Part II

Formal verification of security policies using Frama-C
Security policies verification using Frama-C

Safety properties

- Memory safety: e.g. absence of buffer overflows
- Absence of integer overflows

Error propagation

Analysing the behavior of stream ciphers when a bit in the ciphertext is flipped over the communication channel.

Program equivalence

Verifying the correctness of cryptographic algorithms implementations with respect to a reference implementation – the specification of cryptographic algorithms acts as a reference implementation – code refactoring.

Adherence to side-channel countermeasures

Verifying if the NaCl library attests adherence to side-channel countermeasures.
Case-study: RC4 OpenSSL implementation

RC4

- Symmetric cipher which is structured as two independent blocks
- Its security resides in the strength of the key stream generator
- Informally, at each step, it combines each bit $x_i$ of the input with a bit $k_i$ generated by the key stream generator and produces a bit $y_i$ of the output
Verifying safety properties on RC4 OpenSSL implementation

- Frama-C tool automatically generates proof obligations associated with memory safety and absence of integer overflow
- Automatic proof of safety verification conditions is assisted by assertions like:
  - pre-conditions on the ranges of the inputs
  - loop invariants to reason about the program states before and after loop execution e.g.: / \* @ loop invariant (0 < i <= (len >> 3L)) ... @ \* /
  - simple axioms on ranges of bit-wise operations
- From the annotated program are generated 869 proof obligations
- All the generated proof obligations can be automatically proved using Alt-Ergo and Z3 theorem provers
Error propagation

Stream ciphers property
Flipping one bit of the ciphertext only affects the corresponding decrypted bit.

Encryption algorithm (example)

```c
void encrypt(int input[],int output[],
             int key[],int length){
    int k;
    for(k=0;k<length;k++)
        output[k] = input[k] ^ key[k];
}
```

Informal property description
An error in the position $i$ of the input is not propagated to the positions between $i + 1$ and $length - 1$ of the output.
Noninterference

Informal definition
Public outputs must not depend on the values of the secret inputs.

Security policies

- **Confidentiality** – information cannot flow from high variables to low variables
- **Integrity** – low variables cannot damage information stored in high variables
Formalisation of error propagation using noninterference

- Error propagation can be formalised as a noninterference property;
- **Idea** – high integrity outputs should not depend on low integrity inputs.

**Formalisation of error propagation**

The integrity of the values $\text{output}[i+1..\text{length}-1]$ is not damaged by the value $\text{input}[i]$

- **High integrity:** $\text{input}[0,..,i-1, i+1,..,\text{length}-1]$, $\text{key}[0..\text{length}-1]$, $\text{output}[i+1..\text{length}-1]$, $\text{length}$

- **Low integrity:** $\text{input}[i]$
Formalisation of error propagation (self-composition)

Applying the self-composition (Barthe et. all) definition:

**Pre-condition** High integrity inputs are equal.

```plaintext
for(k=0;k<length;k++)
    output[k] = input[k] ^ key[k];

for(k1=0;k1<length1;k1++)
    output1[k1] = input1[k1] ^ key1[k1];
```

**Post-condition** High integrity outputs are equal.
Programs equivalence (by example)

Informal property
For the same input values, the programs produce the same output values.

Specification

```c
int i=0;
while(i<len) {
    outdata[i]=indata[i] ^ RC4NextKeySymbol(key);
    i++;
}
```

Pre-processing

```c
unsigned char keystream[len];
int i=0;
while(i<len) {
    keystream[i] = RC4NextKeySymbol(key);
    i++;
}
i=0;
while(i<len) {
    outdata[i]=indata[i] ^ keystream[i];
    i++;
}
```

- `RC4NextKeySymbol` is the function which processes the key
- Pre-processing algorithm is a refactored version of the specification
Equivalence by composition

General idea
Capture if programs that are executed from indistinguishable states, terminate in states that are also indistinguishable.

\[
\text{Pre } S_1 = S_3 \\
\text{Post } S_2 = S_4
\]
Applying equivalence by composition to prove program equivalence

Pre-condition: All inputs are equal.

Post-condition: All outputs are equal.
General steps to prove composition properties

- Annotate each loop using the *natural invariants* technique
- Define a lemma to express that the loop invariant specifications are equivalent
- Use the Frama-C framework to automatically discharge all the proof-obligations related with the function behavior

**Remark:** the proof of the lemma can be done using Coq proof assistant.
Side-channel attacks

Side-channel attack
Any attack that takes advantage of observing implementation-specific characteristics of cryptographic algorithms implementations;

Addressed 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;
Minimizing exposure to side-channel attacks

**NaCl cryptographic library countermeasures**

- 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.
Usual operational semantics

\[(P, S_1) \downarrow S_2\]

Extended operational semantics

\[(P, S_1) \downarrow (S_2, M, L)\]

- \(M\) - list of memory locations accessed during program execution;
- \(L\) - list of commands executed by the program during its execution;
Formalisation of side-channel countermeasures (2)

Security definition based on noninterference

Informally
For low-equal initial states, executing two instances of the same program, they must agree on the accessed memory locations and on the executed commands.

\[
\text{assert } S_1 = S_2
\]

\[
\text{assert } L_1 = L_2 \land M_1 = M_2
\]

Property
Low integrity inputs should not interfere with the accessed memory locations neither with the executed commands.
Verifying side-channel countermeasures

Verification using Frama-C

- Transform the original program to include two different kind of lists (as ghost variables):
  - Control list – list containing the evaluation of the conditions of all conditional branches and loops;
  - Array access list – for each array variable is created a list containing the accessed array indexes during program execution;
- Annotate each loop with its natural invariant (must include 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
Part III

CAOVerif: A deductive verification tool for CAO
**CAO language**

**CAO: A Cryptography Aware Language and Compiler**

- Domain specific language to describe cryptographic software
- **Idea** – implementation of cryptographic primitives in a way which is close to the notation used in scientific papers and standards
- Is being developed by Work Package 1 in the CACE project
- C-like language that includes *mathematical constructions* and *operations* commonly used in cryptography as natives data types.

**CAO type system**

<table>
<thead>
<tr>
<th>bool</th>
<th>booleans</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits[n]</td>
<td>bit strings of finite length</td>
</tr>
<tr>
<td>vector of $\tau$</td>
<td>generic one-dimensional container</td>
</tr>
<tr>
<td>struct[$e_1; \ldots; e_n$]</td>
<td>structures</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int</th>
<th>arbitrary precision integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod[p]</td>
<td>residues modulo an integer</td>
</tr>
<tr>
<td>matrix[$n_1,n_2$] of $\tau$</td>
<td>algebraic matrices</td>
</tr>
<tr>
<td>mod[ $\tau &lt; X &gt; / p(X)$ ]</td>
<td>extension fields</td>
</tr>
</tbody>
</table>
**CAOVerif**: A Deductive Verification Tool for CAO

**Requirements**

- Allow the same verification techniques enabled by other verification tools for different languages such as C.
- Simplify the verification of security-relevant properties, providing a higher degree of automation.
- Automatically generate assertions whose validity implies the safeness of CAO programs (such as in Frama-C).

**CAOVerif**

- **CAO-SL** – CAO specification language – is mostly inspired by ACSL.
- Our tool translates annotated CAO programs into Jessie input language.
- Jessie generates the proof-obligations using the Why framework.
- Generated proof-obligations can be proved using a proof assistant (e.g. Coq) or an automatic prover (e.g. CVC3).
Jessie Plug-in

- **Main advantage:** little effort is required to develop deductive verification tools based on the Jessie plug-in (e.g. Frama-C and Krakatoa)
- **Input language** – simple typed imperative language which includes logical features such as *first-order logical constructions*:
  - logic datatypes
  - logic functions over these datatypes
  - axioms to reason over the properties of these functions
- Logical features can be used to “extend” the semantics of the target language (Jessie) of our translator.
The development of a new deductive verification tool based on the Jessie plug-in typically implies:

1. translation of the source annotated programs into (extended) Jessie input language

2. together with the adequate axiomatization of the source language datatypes in first-order logic

Remark: The axiomatic model of these datatypes should be optimized to allow us proving some representative examples: requires a compromise between simplicity and expressiveness.
CAOVerif Architecture

Parsing → Type-Checking → Cao2Jessie → Jessie → Why

- CAO annotated program
- AXIOMATIC MODEL of CAO TYPE SYSTEM
- JESSIE translation
- Jessie Plug-in

- VCGen
- Proof Obligations
- Automatic provers (CVC, Z3,...)
- Proof assistants (Coq, PVS,...)
CAO to Jessie

Translation

- Design of a Jessie model that captures the semantic of CAO programs
- Essentially focused on the design of the axiomatic model of the CAO types in first-order logic
- For each type (not supported by Jessie) it includes its axiomatic model in first-order logic

Axiomatic model of the CAO types

For each type we include:

- logical type (to translate CAO types)
- set of logical functions (to translate the operations on these types)
- set of axioms (to reason about CAO programs)
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 like prescribed;

First, one should identify the security relevant properties that might be addressed in the context of cryptographic software;

We demonstrate that these policies can be formalised and verified using tools such as Frama-C framework: safety properties, error propagation, programs equivalence, etc;

We develop a deductive verification tool for CAO which provides a higher degree of automation in the verification of cryptographic primitives.
Conclusions and Future work

Future work

- Establishing CA0Verif correctness – in related work we are working on an operational semantics for CA0, which we will later use to establish a correctness result for our VCGen; \(^2\)
- We mean correctness by: a CA0 program proved correct in Jessie must be proved correct in CA0 too (soundness of the VCGen);
- Problems:
  - Although the control structures of both programs are similar, the type theory on both languages differs;
  - There isn’t an operational semantic of Jessie language – Jessie is not an executable language;
  - There isn’t any formalisation of Jessie VCGen;

\(^2\)We will rely on the certification of the Jessie plug-in of the Frama-C framework
Future work

Until now, we have:
Publications

J. Bacelar Almeida, M. Barbosa, J. Sousa Pinto and B. Vieira
*Deductive verification of cryptographic software*

J. Bacelar Almeida, M. Barbosa, J. Sousa Pinto and B. Vieira
*Formal verification of side-channel countermeasures using self-composition*

M. Barbosa, J-C. Filliâtre, J. Sousa Pinto and B. Vieira
*An open-source deductive verification platform for cryptographic software*