Verifying C Cryptographic Protocol Implementations by Symbolic Execution

Supervisors: Cristian Ene∗ Laurent Mounier†

October 2015

1 Context

Many applications rely on complex cryptographic protocols for communicating over the insecure Internet (e.g., online banking, electronic commerce, social networks, etc). The C programming language is largely used in writing cryptographic software. Both the design of protocols and their C implementation are error prone. Recent years have seen a real progress in the formal verification of cryptographic protocols as illustrated by the development of several tools both in the symbolic model (Proverif [Bla01], AVISPA [ABB+05], Hermes [BEJ+05]) and in the computational model (CryptoVerif [Bla08], CertiCrypt [BGZB09], EasyCrypt [BGHB11]). There remains however a large gap between what we verify (the protocol usually described in a process algebra, as pi calculus for example) and what we rely on (the implementation which is usually done in a «real» language, like C).

The need to verify the code is now well recognized, but only a few recent works try to propose solutions.

One of the first attempts at cryptographic verification of C code is the CSur ([GLP05]): one extracts from a C program a set of Horn clauses that are then solved using a theorem prover. Some limits of this approach: 1) it can be used to prove secrecy, but it is not clear how one can apply the tool to handle authentication properties (the order of instructions is completely ignored); 2) the results that are obtained are sound only in the symbolic (Dolev-Yao) model of cryptography.

A second line of research, based on symbolic execution ([Kin76]), is that of [CM11], [CM12]; [CM11] extends the KLEE test-generation tool ([CDE08]) by treating certain concrete functions, like cryptographic primitives, as symbolic functions, that is, their execution is avoided, and their behaviour is modelled via rewriting rules. However, their work does not extend the class of properties supported by KLEE, in particular they do not take into account how the inputs provided by an adversary depend on the knowledge learnt by the same adversary. [CM12] extends the previous work with a tainting mechanism that tracks information flows of data, but this work suffers from several limitations:

∗Cristian.Ene@imag.fr
†Laurent.Mounier@imag.fr
it cannot handle authentication properties, and it is done only in the Dolev-Yao model.

Another work based on symbolic execution is that of [AGJ11]. The authors first use the CIL ([NMRW02]) tool to compile the C program down to CVM (a simple stack-based instruction language). Then, they symbolically execute the CVM program in order to eliminate memory accesses and destructive updates, and get an equivalent program in an intermediate model language (IML), actually a version of applied pi calculus enriched with bitstring manipulation operations. Then, they abstract the bitstring operations, and get a process in the fragment of applied pi calculus supported by ProVerif (Proverif [Bla01]), and hence they are able to use ProVerif for checking some security properties in the symbolic model. In a subsequent work [AGJ12], the authors changed the last part of their approach: they translate the extracted IML program to a CryptoVerif protocol description, such that a successful verification with CryptoVerif implies the security of the original C implementation in the computational model. One limitation of this work is that their current method and prototype can analyze only a single execution path, so it is limited to programs/protocols with no significant branching.

2 Goals

The goal of this thesis is to develop a method and a prototype tool in order to verify security properties of C code that uses cryptographic primitives. The objective is to overcome the limitations of the existing methods. More specifically, the aim is to develop a method that starting with a C program (possibly with branching and loops), first extracts an equivalent program in a higher level model, and then use an existing tool in order to verify for security properties.

The work can be organized as follows:

- compile the C program down to a simpler language (CVM or LLVM [LA04] for example)
- define an appropriate intermediate language (denoted here by IL) and give an automatic method based on symbolic execution that allows to extract an equivalent IL program from the compiled version of the initial C program
- give an automatic method to extract an equivalent model EM from the above obtained IL program, such that EM can be used as input for an existing tool (if possibly, EasyCrypt or CryptoVerif in order to get computational soundness)
- implement the method in a prototype tool.

3 Bibliography

References

[ABB+05] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, P Hankes Drielsma, Pierre-


