First Building Blocks for Implementations of Security Protocols Verified in Coq

Reynald Affeldt[1] and Kazuhiko Sakaguchi[1,2]

[1]National Institute of Advanced Industrial Science and Technology
[2]University of Tsukuba

Summary In this presentation, we would like to report on recent case studies of verification in Coq using Separation logic: publicize verified assembly programs for multi-precision arithmetic [2] and report on our progress about verification of network packet parsing written in C (progress since [4]). Our presentation may be of interest to the Coq community because our case studies can serve as informative benchmarks to others. There is indeed a lot of research about verification in Coq of low-level programs using Separation logic, with contributions ranging over encoding techniques (e.g., [8]), tactics to shorten proof scripts (e.g., [6]), or automation (e.g., [7]). Conversely, we hope this community can help us overcome technical issues and organizational challenges we are facing in our way to improve the implementation of security protocols with formal verification.

Our General Approach and Recent Case Studies Our goal is to build low-level programs completely inside the Coq proof-assistant: provide assembly and C programs as Coq inductive types, specify and verify them using variants of Separation logic, and pretty-print them afterward to concrete syntax. As of today, such an approach to program development is not realistic in general, but for critical pieces of code it may turn out to be reasonable in practice [9].

A Library of Multi-precision Arithmetic Functions In [2], we verify several assembly programs for signed multi-precision arithmetic, for a total of 313 lines of MIPS assembly spread over 25 functions: basic initialization functions, signed multi-precision halving, doubling, signed addition, subtraction, various comparisons, etc. The largest function is an implementation of the binary extended gcd algorithm that given two integers a and b, finds x and y such that $\gcd(a, b) = ax + by$.

Verified assembly is a must-have for multi-precision arithmetic because it is used in software such as cryptosystems that call for both performance and security. The implementations we verify are all the more realistic that they manipulate multi-precision integers implemented similarly to the celebrated GNU Multi-Precision library.

Our verification of arithmetic functions is carried out by establishing a simulation between pseudo-code (using arbitrary large integers) and assembly code (using multi-precision integers): the pseudo-code can therefore be seen as a concise specification of the assembly.

Network Packet Parsing written in C In [4], we propose an original encoding of C programs with an application to the parsing of network packets. The novelty is that we use dependent types to achieve a so-called *intrinsic encoding*: only well-typed C programs can be formally modeled. The motivation for choosing such a case study is that parsing of network packets is an important source of bugs in the implementation of security protocols (see, e.g., OpenSSL). Concretely, we have been verifying around 100 lines of C that parse initialization packets for the TLS protocol, using code taken from an existing implementation of TLS (namely, PolarSSL). We also provide pretty-printing so that verified code can be retrofitted inside the original software.

Although C seems to be more structured and expressive than assembly, interactive formal verification using Separation logic turns out to be more difficult. Establishing the absence of overflows when performing arithmetic makes proof tedious and the semantics of C, being much more complicated than the semantics of assembly, leads to large proof terms that slow down type-checking.

A Unifying Case Study: The Implementation of Security Protocols Above case studies can be seen as building blocks of more interesting programs. The implementation of a security protocol is a typical example of security-critical program that mixes C functions (to process network packets) with cryptoschemes (implemented with arithmetic functions). For example, let us consider the ElGamal cryptoscheme. Key generation consists in choosing a prime p, a generator $\langle g \rangle = \mathbb{Z}_p^*$, and a random number $1 \leq a \leq p-2$, and computing $k = g^a (\bmod\, p)$. To encrypt a message $m \in \mathbb{Z}_p$, generate a random number $1 \leq r \leq p-2$, compute $w = g^r (\bmod\, p)$ and $x = m \cdot k^r (\bmod\, p)$: (w, x) is the ciphertext. Its decryption is $x \cdot w^{-a} = m \cdot k^r \cdot (g^r)^{-a} = m \cdot (g^a)^r \cdot (g^r)^{-a} = m\,(\bmod\, p)$. In fact, we already have several pieces of verified code for ElGamal: modular exponentiation in the encryption is typically implemented using the Montgomery algorithm (see [3] for multiplication, [1] for exponentiation), multiplicative inverses can be computed using the binary extended gcd algorithm from [2], and [5] provides a cryptographically-secure pseudo-random number generator. In the implementation of security protocols, the role of C functions would be to break plaintexts into blocks and concatenate decrypted blocks, or to implement a padding scheme, which is an activity akin to network packet processing as we experiment in [4].

Work in Progress We are currently in the process of completing the case study of [4] (network packet parsing written in C). We are facing performance issues due to the size of proof terms and still need to enrich our base libraries with technical lemmas to deal with overflows of finite-size integers.

We are also working on providing pretty-printing inside Coq for a subset of C programs. (We already experimented with a similar facility to produce a working pseudo-random number generator in assembly [5].)

To verify implementations of security protocols as described above, we are working on extending our libraries to verify programs that mix C and (potentially inlined) assembly code. The simulation we established in [2] between pseudo-code and assembly is actually defined using two instances of the same set of modules for Hoare logic. It should be possible to reuse this work to enable a simultaneous use of assembly and C.

References

[1] A Library for Formal Verification of Low-level Programs. Coq scripts. Available at `http://staff.aist.go.jp/reynald.affeldt/coqdev`

[2] R. Affeldt. On Construction of a Library of Formally Verified Low-level Arithmetic Functions. To appear in Innovations in Systems and Software Engineering. Springer, 2013

[3] R. Affeldt, N. Marti. An Approach to Formal Verification of Arithmetic Functions in Assembly. In ASIAN 2006, LNCS 4435:346–360. Springer, 2008

[4] R. Affeldt, N. Marti. Towards Formal Verification of TLS Network Packet Processing Written in C. In PLPV 2013:35–46. ACM, 2013

[5] R. Affeldt, D. Nowak, K. Yamada. Certifying Assembly with Formal Security Proofs: the Case of BBS. Science of Computer Programming, 77(10–11):1058–1074. Elsevier, 2012

[6] J. Bengtson, J. B. Jensen, L. Birkedal, Charge! - A Framework for Higher-Order Separation Logic in Coq. In ITP 2012, LNCS 7406:315–331. Springer, 2012

[7] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In PLDI 2011:234–245. ACM, 2011.

[8] J. B. Jensen, N. Benton, A. Kennedy. High-level separation logic for low-level code. In POPL 2013:23–25. ACM, 2013

[9] G. Klein. From a Verified Kernel towards Verified Systems. In APLAS 2010, LNCS 6461:21–33. Springer, 2010