Science of Computer Programming 77 (2012) 1058–1074

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Certifying assembly with formal security proofs: The case of BBS☆

Reynald Affeldt*, David Nowak, Kiyoshi Yamada[1]

National Institute of Advanced Industrial Science and Technology (AIST), Central 2, 1-1-1 Umezono, Tsukuba, Ibaraki, 305-8568, Japan

ARTICLE INFO

Article history:
Available online 13 July 2011

Keywords:
Hoare logic
Assembly language
Coq
PRNG
Provable security

ABSTRACT

With today's dissemination of embedded systems manipulating sensitive data, it has become important to equip low-level programs with strong security guarantees. Unfortunately, security proofs as done by cryptographers are about algorithms, not about concrete implementations running on hardware. In this article, we show how to perform security proofs to guarantee the security of assembly language implementations of cryptographic primitives. Our approach is based on a framework in the Coq proof assistant that integrates correctness proofs of assembly programs with game-playing proofs of provable security. We demonstrate the usability of our approach using the Blum–Blum–Shub pseudorandom number generator, for which an MIPS implementation for smartcards is shown cryptographically secure.

1. Introduction

With today's dissemination of embedded systems manipulating sensitive data, it has become important to equip low-level programs with strong security guarantees. However, despite the fact that most security claims implicitly assume correct implementation of cryptography, this assumption is never formally enforced in practice. The main problem of formal verification of embedded cryptographic software is that, in the current state of research, formal verification remains a major undertaking:

(a) Most cryptographic primitives rely on number theory and their pervasive usage calls for efficient implementations. As a result, we face many advanced algorithms with low-level implementations in assembly language. This already makes formal proof technically difficult.
(b) Security guarantees about cryptographic primitives is the matter of *security proofs*, as practiced by cryptographers. In essence, these proofs aim at showing the security of cryptographic primitives by reduction to computational assumptions. Formal proofs of such reductions also involve probability theory or group theory.

In fact, formal verification of embedded cryptographic software is even more challenging in that it requires a formal integration of (a) and (b). To the best of our knowledge, no such integration has ever been attempted so far.

In this article, we address the issue of formal verification of cryptographic assembly code with security proofs. As pointed out above, formal verification of cryptographic assembly code and formal verification of security proofs are not the same matter, even though both deal with cryptography. As an evidence of this mismatch, one can think of a cryptographic function such as encryption: its security proof typically relies on a high-level mathematical description, but when laid down in terms

☆ A preliminary version of this work appeared in the proceedings of the 9th International Workshop on Automated Verification of Critical Systems (Affeldt et al., 2009) [1].

* Corresponding author.
E-mail address: reynald.affeldt@aist.go.jp (R. Affeldt).
[1] Present address: Lepidum Co., Ltd., Village Sasazuka III Bldg 6F, 1-30-3 Sasazuka, Shibuya-ku, Tokyo, 151-0073, Japan.

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.07.003

of assembly code such a function exhibits restrictions due to the choice of implementation. We are therefore essentially concerned about the integration of these two kinds of formal proofs. We do not question here the theoretical feasibility of such an integration; rather, we investigate its practical aspects when formal verification is carried out within a proof assistant based on proof theory.

Indeed, proof assistants based on proof theory, such as Coq [9] or HOL, can be regarded as privileged tools when it comes to formal verification of security properties. They implement higher order logics that are expressive enough to model and reason about advanced mathematics as well as precise computation models, and their trusted computing base being small and well-understood provides adequate reliability. With such tools, formal verification is by default performed interactively (by means of *tactics*) but interaction can be automated and intermediate results can be aggregated (in the form of libraries of *lemmas*) to facilitate other formal verifications. As a consequence, various frameworks for formal verification of cryptography using proof assistants based on proof theory have already been proposed: on the one hand, [2,18] for cryptographic assembly code, and on the other hand, [4–6,20] for security proofs. However, it is not clear how to connect them in practice, or, in other words, how the formal security proof for a cryptographic primitive relates to its formally verified implementation. Whatever connection is to be provided between two such frameworks, it has to be developed in a clear way, both understandable by cryptographers and implementers, and in a reusable fashion, so that new verification efforts can build upon previous ones.

Our main contribution is to propose a concrete approach, supported by a reusable formal framework on top of the Coq proof assistant [9], for verification of assembly code together with security proofs. As a concrete evidence of usability, we formally verify a pseudorandom number generator written in assembly for smartcards with a proof of unpredictability. This choice of application is not gratuitous: this is the first step before verifying more cryptographic primitives, since many of them actually rely on pseudorandom number generation. To achieve our goal, we extend and integrate two existing frameworks developed for the Coq proof assistant: one for formal verification of assembly code [2], and another for formal verification of cryptographic primitives [20]. More precisely, our technical contributions consist in the following:

- We propose an integration in terms of *game-playing* [24], a popular setting to represent security proofs. We introduce a new kind of game transformation to serve as a bridge between assembly code and algorithms as dealt with by cryptographers. This allows for a clear integration, that paves the way for a modular framework, understandable by both cryptographers and implementers.
- We extend the formal framework for assembly code of [2] to connect with the formal framework for security proofs of [20]. Various technical extensions are called for, that range from the natural issue of encoding mathematical objects such as arbitrarily large integers into computer memory, to technical issues such as composition of assembly snippets to achieve verification of large programs. All in all, it turns out that it is utterly important to provide efficient ways to deal with low-level details induced by programs being written in assembly. Here, we explain in particular how we deal with arbitrary jumps in assembly. Concretely, we provide a formalization of the proof-carrying code framework of [23], that allows us to verify assembly with jumps through standard Hoare logic proofs.
- We provide the first assembly program for a pseudorandom number generator that is formally verified with a security proof. The generator in question is the Blum–Blum–Shub pseudorandom number generator (hereafter, BBS) [8] that we implement in the SmartMIPS assembly.

Outline. In Section 2, we introduce the BBS algorithm and provide an assembly implementation. In Section 3, we explain how we integrate formally proofs of functional correctness for assembly code with game-based security proofs. In Section 4, we explain our formalization of the proof-carrying code framework of [23], that facilitates formal proof of functional correctness of assembly code. In Section 5, we explain the formal security proof of BBS in assembly, from its proof of functional correctness to the implementation step that enables integration with the security proof from [21]. In Section 6, we validate our formalization by producing automatically a SmartMIPS binary and experimenting its execution. In Section 7, we comment on technical aspects of the Coq formalization. In Section 8, we comment on related work, and conclude and comment on future work in Section 9.

2. The BBS pseudorandom number generator

2.1. The BBS algorithm

The security of the Blum–Blum–Shub pseudorandom number generator [8] is based on mathematical properties of integers modulo. We recall some basic notions of number theory: an integer is a quadratic residue modulo m if it is congruent to a square modulo m; the Legendre of an integer (modulo a prime number) is $+1$ if it is a quadratic residue, and -1 otherwise; the Jacobi of an integer modulo m is the product of its Legendres for each prime factor of m. BBS exploits the *quadratic residuosity problem*: the problem of deciding for $\mathbb{Z}_m^*$ (the multiplicative group of integers modulo m), where m is the product of two distinct odd primes, whether elements with Jacobi $+1$ are in

$$QR_m \stackrel{def}{=} \{x \in \mathbb{Z}_m^* | \exists y \in \mathbb{Z}_m^*. y^2 \pmod m = x\}.$$

The *quadratic residuosity assumption* is the assumption that this problem is intractable ([10], ch. 6). BBS exploits the quadratic residuosity assumption in the particular case of m being a Blum integer, i.e., the product of two distinct odd primes congruent