Certifying Assembly with Formal Cryptographic Proofs: the Case of BBS[*]

Reynald Affeldt, David Nowak and Kiyoshi Yamada

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 paper, 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 (BBS) pseudorandom number generator, for which a 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 *cryptographic 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 addition, formal verification of embedded cryptographic software is even more challenging in that it requires a formal integration of (a) and (b). In fact, to the best of our knowledge, no such integration has ever been attempted so far.

In this paper, we address the issue of formal verification of cryptographic assembly code with cryptographic proofs. As pointed out above, formal verification of cryptographic assembly code and formal verification of cryptographic proofs are not the same matter, even though both deals 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 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. Indeed, various frameworks for formal verification of cryptography using proof assistants based on proof theory already exist ([AM06, MG07] for cryptographic assembly code, [ATM07, BBU08, BGZ09, Now07] for cryptographic proofs), but it is not clear how to connect them in practice. Whatever connection is to be provided, it has to be developed in a

---

[*]This work appeared in the proceedings of AVoCS 2009.

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, for verification of assembly code together with cryptographic 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 integrate two existing frameworks: one for formal verification of assembly code, and another for formal verification of cryptographic primitives. More precisely, our technical contributions consist in the following:

• We propose an integration in terms of *game-playing* [Sho04], a popular setting to represent cryptographic 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 [AM06] to connect with the formal framework for cryptographic proofs of [Now07]. 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 [SU07], that allows us to verify assembly with jumps through standard Hoare logics proofs.

• We provide the first assembly program for a pseudorandom number generator that is formally verified with a cryptographic proof. The generator in question is the Blum-Blum-Shub pseudorandom number generator [BBS86] that we implement in the SmartMIPS smartcard assembly.

Alternative Approaches and Related Work Our approach is oriented towards practical application, and this goal includes formal verification of hand-written assembly. For this purpose, extension of [AM06] is appropriate because it already provides much material for reasonably short proof scripts. One may think of alternative approaches to verify cryptographic assembly code, such as proof-producing compilation or refinement from a functional specification. However, a proof-producing compiler (such as the one of [MSG09]) would need to be extended with custom support for cryptography-specific instructions in order to produce efficient code. Regarding the approach by refinement from a functional specification, the application to cryptographic assembly code in [MG07] does not seem to lead to shorter proof scripts and compact assembly code. Addressing these issues would still be not enough for our overall goal, for HOL (the proof assistant used in [MSG09] and [MG07]) lacking a framework for formal cryptographic proofs.

The two existing frameworks ([AM06] and [Now07]) that we integrate in this paper turn out to be a good fit for they are both based on shallow encodings. On the one hand, shallow encoding is used in [AM06] to encode Hoare logic assertions, and on the other hand, it is used in [Now07] to represent games. Therefore, algorithms written as Coq functions can simply appear in Hoare logic assertions, making for an easy integration. In contrast, games in [BBU08, BGZ09] are represented as deep-encoded code. In addition, our use-case directly relies on properties of arithmetic (including an encoding of the quadratic residuosity problem) an originality of [Now08].

Outline In Sect. 2, we introduce the BBS algorithm and provide an assembly implementation. In Sect. 3, we explain how we integrate formally proofs of functional correctness for assembly code with game-based cryptographic proofs. In Sect. 4, we explain our formalization of the proof-carrying code framework of [SU07], that facilitates formal proof of functional correctness of assembly code. In Sect. 5, we explain the formal proof of functional correctness of BBS and the lemmas relevant to the integration with its cryptographic proof. In Sect. 6, we comment on technical aspects of the Coq formalization. We conclude and comment on future work in Sect. 7.

2 The BBS Pseudorandom Number Generator

2.1 The BBS Algorithm

The Blum-Blum-Shub pseudorandom number generator [BBS86] (hereafter, BBS) exploits the *quadratic residuosity problem*. This problem is to decide whether integers have square roots in modular arithmetic. This is believed to be intractable for multiplicative group of integers modulo m where m is the product of two distinct odd primes. BBS exploits the quadratic residuosity problem in the particular case of m being a Blum integer, i.e., the product of two distinct odd primes congruent to 3 modulo 4.

Here follows an implementation of BBS as a Coq function. It performs iteratively squaring modulo and outputs the result of parity tests. The input is the desired number of pseudorandom bits (len) and a random seed ($seed$) for initialization. $\mathbb{Z}_m^*$ is the multiplicative group of integers modulo m and QR_m is the set of quadratic residues modulo m.

$$\mathsf{bbs}(len \in \mathbb{N}, seed \in \mathbb{Z}_m^*) =_{def} \mathsf{bbs\_rec}(len, seed^2)$$
$$\mathsf{bbs\_rec}(len \in \mathbb{N}, x \in QR_m) =_{def} \mathbf{match}\ len\ \mathbf{with}$$
$$|\ 0\ \Rightarrow\ []$$
$$|\ len' + 1\ \Rightarrow\ \mathsf{parity}(x) :: \mathsf{bbs\_rec}(len', x^2)$$
$$\mathbf{end}$$

BBS is one of the rare pseudorandom number generators that is *cryptographically secure*, i.e., it passes all polynomial-time statistical tests (no polynomial-time algorithm can distinguish between an output sequence of the generator and a truly random sequence). This strong property is not required of most applications of pseudorandom numbers, except cryptography. In practice, BBS can be proved *left-unpredictable* (hereafter, "unpredictable") under the assumption that the quadratic residuosity problem is intractable (this is equivalent to prove that BBS passes all polynomial-time statistical tests [Yao82].)

2.2 Implementation of BBS in Assembly

The assembly code bbs_asm in Fig. 2.2 implements BBS. It is written with MIPS instructions (actually, we use SmartMIPS, a superset of MIPS32 with additional instructions for smartcards [Mips]). It consists

```
bbs_asm =def
0:      addiu i gpr_zero 0₁₆          (* init counter for outer loop *)
1:      addiu L l 0₁₆                   (* init pointer to result *)
2:      beq i n 240                     (* repeat n times *)
3:        addiu j gpr_zero 0₁₆        (* init counter for inner loop *)
4:        addiu w gpr_zero 0₁₆       (* init word of temporary storage *)
5:        beq j thirtytwo 236          (* repeat 32 times *)
6:          mul_mod k x x m ...         (* compute X² (mod M) *)
222:         lw w' 0₁₆ x              (* load least significant word *)
223:         andi w' w' 1₁₆            (* extract parity bit *)
224:         sllv w' w' j              (* shift parity bit to jth position *)
225:         cmd_or w w w'        (* store parity bit in temporary storage *)
226:         addiu j j 1₁₆            (* increment inner loop counter *)
227:         jmp 5                     (* end of the inner loop *)
228:       sw w 0₁₆ L          (* store the last 32 parity bits in memory *)
229:       addiu L L 4₁₆               (* increment pointer to result *)
230:       addiu i i 1₁₆             (* increment outer loop counter *)
231:       jmp 2                      (* end of the outer loop *)
232:
```

Figure 1: The Blum-Blum-Shub pseudorandom number generator in assembly

of a loop with a nested loop. Each iteration of the nested loop produces one word of pseudorandom bits by performing a square modulo, extracting the parity bit (of the least significant word), and storing this bit in a temporary word of storage in an appropriate position using bitwise operations. These temporary

words of storage are then stored in memory contiguously by the outer loop so as to produce a pseudorandom number. The names of registers (in italic font) are parameters; only the null register `gpr_zero` is hardwired in the program. Magic numbers are indexed with their length in bits (e.g., 0_{16} stands for 0 represented as a half-word). `mul_mod` is an inlined assembly program explained in the next section.

2.3 Implementation of Modular Multiplication in Assembly

We implement multi-precision square modulo using the Montgomery multiplication [Mon85]. This is not the fastest way to implement multi-precision square modulo but, still, this is reasonable: like the natural multi-precision multiplication/division, it has a quadratic complexity. Moreover, we already have a formal proof for an optimized version of the Montgomery multiplication [AM06], whereas, to the best of our knowledge, such a formal proof for multi-precision division does not exist yet.

Using Montgomery, modular multiplication is performed as follows. Given three k-word integers M, X, Y, the Montgomery multiplication computes a $k + 1$-word integer Z such that $\beta^k Z = X.Y \pmod{M}$ and $Z < 2M$ ($\beta = 2^{32}$). This is almost a multiplication modulo except for the parasite value β^k and because $Z \not< M$ in general. To turn it into a genuine multiplication modulo, one needs (1) an additional subtraction to reduce Z by M when necessary and (2) two passes to eliminate the parasite value. The second pass requires as an additional input a k-word $A = \beta^{2k} \pmod{M}$; given Z such that $\beta^k Z = X.Y \pmod{M}$, it suffices to compute Z' such that $\beta^k Z' = Z.A \pmod{M}$: if M is odd (this is generally the case for cryptographic applications), one obtains as desired $Z' = X.Y \pmod{M}$.

The assembly code `mont_mul_strict_init` in Fig. 2.3 implements the Montgomery multiplication extended with comparison and subtraction. It makes use of the functions `montgomery` (the function verified in [AM06]),

```
mont_mul_strict_init =_def
6:      multi_zero ext k Z z                        (* output initialization *)
13:     mflhxu gpr_zero                             (* multiplier inialization *)
14:     mthi gpr_zero
15:     mtlo gpr_zero
16:     montgomery k alpha x y z m one ext int X Y M Z quot C t s
54:     beq C gpr_zero 79              (* is the output k + 1-word long? *)
55:       addiu t t 4_16
56:       sw C 0_16 t
57:       addiu ext k 1_16
58:       multisub ext one z m z M int quot C Z X Y X
78:       jmp 114
79:     multi_lt_prg k z m X Y int ext Z M
91:     beq int gpr_zero 94      (* is the output bigger than the modulus? *)
92:       skip
93:       jmp 114
94:       multisub k one z m z ext int quot C Z X Y X
114:
mul_mod =_def
6:      mont_mul_strict_init k alpha x x y m one ext int X B2K Y M quot C t s
114:    mont_mul_strict_init k alpha y b2k x m one ext int X B2K Y M quot C t s
222:
```

Figure 2: The Montgomery multiplication extended with comparison and subtraction

(in-place) subtraction `multi_sub` (derived from [AM06]), multi-precision comparison `multi_lt` and an initialization function `multi_zero` (see [Code] for the details). The assembly code `mul_mod` in Fig. 2.3 perform a square modulo by using twice `mont_mul_strict_init`, provided we assume that the register $b2k$ points to the pre-computed k-word $A = \beta^{2k} \pmod{M}$.

3   Game-based Proofs for Assembly

Cryptographic proofs usually apply to algorithms without any consideration for implementation. In order to prove unpredictability directly on assembly code, we propose to lift a standard definition borrowed from game-playing [Sho04]. Game-playing is a methodology to write cryptographic proofs that are easier to verify; it lends itself well to formalization [ATM07, BBU08, BGZ09, Now07]. A security property is modeled as a *game* (a program) to be solved by an attacker, the latter being modeled as some probabilistic procedure. A cryptographic proof consists in showing that any attacker has only little advantage over a random player, by (1) stating the security property for the cryptographic primitive to be verified, and (2) reducing it to a computational assumption through *game transformations*.

Regarding unpredictability for a function f, the game unpredictability(f) is defined as follows [Now08]: a *seed* is picked at random in the set of seeds G ($\mathbb{Z}_n^*$ in the case of BBS); a sequence of bits $[b_0, \ldots, b_{len}]$ is computed by f; this sequence, deprived of its first bit b_0, is passed to the attacker A; the latter returns its guess $\widehat{b_0}$. The result of the game is whether the guess is right or not. To define unpredictability for assembly code, one needs to lift the previous definition because it applies to mathematical functions without any consideration for their implementation. This makes a difference because, contrary to mathematical functions, assembly code does not work as intended for arbitrary input, due to restrictions imposed by the choice of implementation. The basic idea is thus to extract from the assembly code its semantics in terms of a mathematical function and to inject it into the definition of unpredictability: unpredictability_assembly(c) $=_{def}$ unpredictability($[\![c]\!]$). For $[\![c]\!]$ to be well-defined, the assembly code c has to be deterministic and terminating, and, more importantly, one needs to make clear under which restrictions the assembly code behaves as intended (the correctness property).

The advantage of the lifting explained above is that it makes clear how to organize formal verification of assembly code with cryptographic proofs. Games for assembly connect to standard games through *implementation steps*, that are justified formally by ensuring determinism, termination, and correctness. Since implementation steps come in addition to the other game transformations [Sho04], this makes it easier to develop a formal framework for verification of assembly with cryptographic proofs: pick up a formal framework for game-based proofs and a formal framework for assembly, and add the machinery for implementation steps.

4   Verification of Functional Correctness of Assembly

To perform cryptographic proofs of an implementation, we need in particular to prove its functional correctness. This is technically difficult for assembly because handling of jumps results in non-standard logics, usually verbose, and thus less practical than standard Hoare logic. To overcome this difficulty, we formalize the proof-carrying code framework of [SU07] that provides not only a compositional operational semantics and Hoare logic for assembly with jumps, but also shows that derivations for this non-standard operational semantics and this Hoare logic can be obtained from standard operational semantics and standard Hoare logic by compilation. The following is an overview of our formalization of [SU07]; it includes a formalization of standard Separation Logic that is actually a revision of [AM06] recalled for the sake of self-containedness.

4.1   Operational Semantics

Formalization of States A state is a pair of a *store* and a *heap*: $state =_{def} store \times heap$. A store is a collection of registers containing integers of finite size. Let int_n be the type of machine integers encoded with n bits. Most registers contain values of type int_{32} (the exception is the *extended accumulator* of type int_n with $n \geq 8$). We have the following notations: $[\![r]\!]_s$ is the value of register r in store s; $s\{v/r\}$ is the store resulting from updating register r with value v in store s. A heap is a finite map from locations to integers of type int_{32}. The heap is tailored to word-accesses because most memory accesses in our applications are word-aligned. We have the following notation: $h[l]$ is the contents of location l of the heap h; it is `None` when the location is undefined.

States are extended with a *label* (that represents the value of the program counter of the instruction being currently executed) and can be *error states* (because some instructions may trap). We distinguish error states using an `option` type, hence the definition of labelled states: $lstate =_{def}$ `option` $(label \times state)$.

One-step, Non-branching Instructions The semantics of non-branching MIPS instructions is a predicate noted $s - i \longrightarrow s'$ where i is a MIPS instruction, s (resp. s') is the state before (resp. after) its execution. When formalizing the semantics of instructions, we need to express conditions such as word-alignment, absence of arithmetic overflow, etc. These conditions require manipulations such as sign-extending int_{16} integers to int_{32} integers, checking for divisibility by 4, etc. For this purpose, we introduce various operators: $(v)_{int_{16} \to int_{32}}$ sign-extends the value v from 16 to 32 bits, $(v)_{int_{32} \to \mathbb{N}}$ interprets the value v as an unsigned integer, etc.

Figure 3 illustrates the semantics of MIPS instructions with the rules for the instruction `lw` ("load word"). There are two rules depending on whether the memory access is word-aligned and the accessed location is defined. (The notation $+_h$ is the addition of finite-size integers.)

$$\frac{\left([\![base]\!]_s +_h (off)_{int_{16} \to int_{32}} \right)_{int_{32} \to \mathbb{N}} = 4 \times p \quad h[p] = \texttt{Some } z}{\texttt{Some } (s, h) - \texttt{lw } rt \ off \ base \longrightarrow \texttt{Some } (s\{z/rt\}, h)} \quad \text{exec0\_lw}$$

$$\frac{\forall p. \ \left([\![base]\!]_s +_h (off)_{int_{16} \to int_{32}} \right)_{int_{32} \to \mathbb{N}} \neq 4 \times p \ \lor \ h[p] = \texttt{None}}{\texttt{Some } (s, h) - \texttt{lw } rt \ off \ base \longrightarrow \texttt{None}} \quad \text{exec\_lw\_error}$$

Figure 3: Semantics of `lw`

Big-step Operational Semantics of Assembly with Jumps Following [SU07], an assembly program is formalized as a set of labelled instructions. The latter are either labelled MIPS instructions or jump instructions (unconditional jumps `jmp l` or conditional jumps `cjmp b l`). Conditional jumps comprise MIPS instructions such as `bne` ("branch if not equal"), etc; these instructions are parameterized by conditions noted b. $dom(c)$ is the set of the labels of the instructions of the assembly program c. Labelled instructions are assembled using $\oplus$.

The operational semantics of assembly programs is a predicate noted $s \succ c \twoheadrightarrow s'$ where c is a set of labelled instructions, s (resp. s') is the state before (resp. after) its execution. It is defined inductively by the rules of Fig. 4. These rules are a generalization of [SU07] with more instructions and with error states. The originality of this semantics can be appreciated by looking at the two rules for sequences using $\oplus$; intuitively, they are a mix of the rules for sequence and while-loops of traditional Hoare logic.

4.2 Hoare Logics

The non-standard operational semantics of the previous section gives rise to a non-standard Hoare logic for assemblies with jumps. We now formalize the Hoare logics from [SU07] (actually, extensions known as Separation Logic [Rey02]).

Assertions Properties of states are specified using a shallow-encoding of the logical connectives of Separation Logic, i.e., assertions are functions from states to the type **Prop** of propositions in Coq: $assertion =_{def} store \to heap \to$ **Prop**. The satisfiability of an assertion can depend on the value of the current label: $assn =_{def} label \to assertion$.

Standard Separation Logic A triple in this logic is noted $\{\mathcal{P}\} c \{\mathcal{Q}\}$ where $\mathcal{P}$ and $\mathcal{Q}$ are assertions and c is an assembly program with while-loops instead of jumps. Let us introduce a function that computes the weakest precondition of one-step, non-branching MIPS instructions. Here is an excerpt of this function for the "load word" instruction:

$$\frac{\text{Some } s - i \longrightarrow \text{Some } s'}{\text{Some } (l,s) \succ l:i \twoheadrightarrow \text{Some } (l+1, s')} \qquad \frac{\text{Some } s - i \longrightarrow \text{None}}{\text{Some } (l,s) \succ l:i \twoheadrightarrow \text{None}}$$

$$\frac{l \neq l'}{\text{Some } (l,s) \succ l:\texttt{jmp } l' \twoheadrightarrow \text{Some } (l', s)}$$

$$\frac{[\![b]\!]_s \quad l \neq l'}{\text{Some } (l,s) \succ l:\texttt{cjmp } b\, l' \twoheadrightarrow \text{Some } (l', s)} \qquad \frac{\neg[\![b]\!]_s}{\text{Some } (l,s) \succ l:\texttt{cjmp } b\, l' \twoheadrightarrow \text{Some } (l+1, s)}$$

$$\frac{l \in \text{dom}(c_1)}{\text{Some } (l,s) \succ c_1 \twoheadrightarrow s' \quad s' \succ c_1 \oplus c_2 \twoheadrightarrow s''}{\text{Some } (l,s) \succ c_1 \oplus c_2 \twoheadrightarrow s''} \qquad \frac{l \in \text{dom}(c_2)}{\text{Some } (l,s) \succ c_2 \twoheadrightarrow s' \quad s' \succ c_1 \oplus c_2 \twoheadrightarrow s''}{\text{Some } (l,s) \succ c_1 \oplus c_2 \twoheadrightarrow s''}$$

$$\frac{}{\text{None} \succ c \twoheadrightarrow \text{None}} \qquad \frac{l \notin \text{dom}(c)}{\text{Some } (l,s) \succ c \twoheadrightarrow \text{Some } (l,s)}$$

Figure 4: Big-step Operational Semantics of Assembly with Jumps

$$\mathcal{WP}\ i\ \mathcal{Q} =_{def} \textbf{match } i \textbf{ with}$$
$$\mid \texttt{lw } rt\ off\ base \Rightarrow \lambda s, h.\exists p.\left([\![base]\!]_s +_h ([\![off]\!]_s)_{int_{16} \to int_{32}}\right)_{int_{32} \to \mathbb{N}} = 4 \times p\ \wedge\ \exists z. h[p] = \texttt{Some } z\ \wedge\ \mathcal{Q}\ s\{z/rt\}$$
$$\mid \dots \textbf{ end}$$

Using the above function, the standard separation logic is defined by the following rules:

$$\frac{}{\{\mathcal{WP}\ i\ Q\}\ i\ \{\mathcal{Q}\}} \qquad \frac{\{\mathcal{P}\}\ c_1\ \{\mathcal{R}\} \quad \{\mathcal{R}\}\ c_2\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ c_1\,;c_2\ \{\mathcal{Q}\}} \qquad \frac{\{\lambda s, h.\mathcal{P} \wedge [\![b]\!]_s\}\ c\ \{\mathcal{P}\}}{\{\mathcal{P}\}\ \texttt{while } b\ c\ \{\lambda s, h.\mathcal{P} \wedge \neg[\![b]\!]_s\}}$$

$$\frac{\{\lambda s, h.\mathcal{P} \wedge [\![b]\!]_s\}\ c_1\ \{\mathcal{Q}\} \quad \{\lambda s, h.\mathcal{P} \wedge \neg[\![b]\!]_s\}\ c_2\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\ \{\mathcal{Q}\}} \qquad \frac{\mathcal{P} \to \mathcal{P}' \quad \{\mathcal{P}'\}\ c\ \{\mathcal{Q}'\} \quad \mathcal{Q}' \to \mathcal{Q}}{\{\mathcal{P}\}\ c\ \{\mathcal{Q}\}}$$

This logic is formally proved sound and complete w.r.t. standard big-step operational semantics, where assembly code with while-loops is built out of MIPS instructions (Sect. 4.1), sequences ($c_1\,;c_2$), structured branching (if b then c_1 else c_2), and while-loops (while $b\ c$):

$$\frac{}{\text{None} - c \twoheadrightarrow \text{None}} \qquad \frac{s - c \longrightarrow s'}{s - c \twoheadrightarrow s'} \qquad \frac{s - c_1 \twoheadrightarrow s'' \quad s'' - c_2 \twoheadrightarrow s'}{s - c_1\,;c_2 \twoheadrightarrow s'}$$

$$\frac{[\![b]\!]_s \quad \text{Some } (s,h) - c_1 \twoheadrightarrow s'}{\text{Some } (s,h) - \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \twoheadrightarrow s'} \qquad \frac{\neg[\![b]\!]_s \quad \text{Some } (s,h) - c_2 \twoheadrightarrow s'}{\text{Some } (s,h) - \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \twoheadrightarrow s'}$$

$$\frac{[\![b]\!]_s \quad \text{Some } (s,h) - c \twoheadrightarrow s'}{s' - \texttt{while } b\ c \twoheadrightarrow s''}{\text{Some } (s,h) - \texttt{while } b\ c \twoheadrightarrow s''} \qquad \frac{\neg[\![b]\!]_s}{\text{Some } (s,h) - \texttt{while } b\ c \twoheadrightarrow \text{Some } (s,h)}$$

Separation Logic based on the compositional Hoare logic of [SU07] A triple in this logic is noted $[\mathcal{P}]\ c\ [\mathcal{Q}]$ where $\mathcal{P}$ and $\mathcal{Q}$ are labelled assertions (type *assn*) and c is an assembly program with jumps. We introduce predicate transformers that enforce assertions to be satisfiable for labels inside (resp. outside) a domain: $\mathcal{P}|_d =_{def} \lambda l.\mathcal{P}\ l \wedge l \in d$, $\mathcal{P}|_{\overline{d}} =_{def} \lambda l.\mathcal{P}\ l \wedge l \notin d$. Using above predicate transformers and the above weakest-precondition function, we formalize the rules for the compositional Hoare logic below. This logic is formally proved sound and complete w.r.t. the big-step operational semantics of Sect. 4.1.

$$\frac{}{\left[\begin{matrix} \lambda pc.\lambda s. & pc = l \wedge (\mathcal{P}\ j\ s \vee j = l)\ \vee \\ & pc \neq l \wedge \mathcal{P}\ pc\ s \end{matrix}\right]\ l:\texttt{jmp } j\ [\mathcal{P}]}$$

$$\frac{}{\left[\begin{matrix} \lambda pc.\lambda s. & pc = l \wedge (\neg[\![b]\!]_s \wedge \mathcal{P}\ (l+1)\ s \vee [\![b]\!]_s \wedge (\mathcal{P}j\ s \vee j = l))\ \vee \\ & pc \neq l \wedge \mathcal{P}\ pc\ s \end{matrix}\right]\ l:\texttt{cjmp } b\ j\ [\mathcal{P}]}$$

$$[\mathcal{P}]\,nop\,[\mathcal{P}] \qquad\qquad \left[\begin{array}{l} \lambda pc.\lambda s. \quad pc = l \wedge \mathcal{WP}\,c\,(\mathcal{P}\,(l+1))\,s\,\vee \\ \qquad\qquad\quad pc \neq l \wedge \mathcal{P}\,pc\,s \end{array}\right] l : c\,[\mathcal{P}]$$

$$\frac{[\mathcal{P}|_{\mathrm{dom}(c_1)}]\,c_1\,[\mathcal{P}] \quad [\mathcal{P}|_{\mathrm{dom}(c_2)}]\,c_2\,[\mathcal{P}]}{[\mathcal{P}]\,c_1 \oplus c_2\,[\mathcal{P}|_{\overline{\mathrm{dom}(c_1 \oplus c_2)}}]} \qquad\qquad \frac{\forall l.\mathcal{P}\,l \rightarrow \mathcal{P}'\,l \quad [\mathcal{P}']\,c\,[\mathcal{Q}'] \quad \forall l.\mathcal{Q}'\,l \rightarrow \mathcal{Q}\,l}{[\mathcal{P}]\,c\,[\mathcal{Q}]}$$

4.3 Compilation from Standard Semantics and Hoare Logic

[SU07] shows that derivations for the previous non-standard operational semantics and Hoare logic can also be obtained from standard operational semantics and Hoare logic through compilation. This is a result of interest because it allows us to work with standard operational semantics and Hoare logic (that are more practical to deal with formally) while still being able to recover formal proofs for assembly with jumps (these are the formal proofs that we really want, for example for shipping in a proof-carrying code scenario).

The compilation procedure turns if-then-else's and while-loops into conditional and unconditional jumps. The compilation of program c with while-loops to an assembly program c' with jumps is noted $c\ ^{l}\searrow_{l'}\ c'$ where l (resp. l') is the start (resp. end) label of the compiled program:

$$\frac{}{i\ ^{l}\searrow_{l+1}\ l : i} \qquad \frac{c_1\ ^{l_1+1}\searrow_{l_2}\ c_1' \quad c_2\ ^{l+1}\searrow_{l_1}\ c_2'}{\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ ^{l}\searrow_{l_2}\ l : \mathtt{cjmp}\ b\ (l_1+1) \oplus ((c_2' \oplus l_1 : \mathtt{jmp}\ l_2) \oplus c_1')}$$

$$\frac{c_1\ ^{l}\searrow_{l_1}\ c_1' \quad c_2\ ^{l_1}\searrow_{l_2}\ c_2'}{c_1 ; c_2\ ^{l}\searrow_{l_2}\ c_1' \oplus c_2'} \qquad \frac{c\ ^{l+1}\searrow_{l_1}\ c'}{\mathtt{while}\ b\ c\ ^{l}\searrow_{l_1+1}\ l : \mathtt{cjmp}\ (\neg b)\ (l_1+1) \oplus (c' \oplus l_1 : \mathtt{jmp}\ l)}$$

Through compilation, derivations of operational semantics can be compiled from the standard one to the non-standard one of Sect. 4.1, and, similarly, proofs in Separation Logic can be compiled from the standard one to the non-standard one of Sect. 4.2:

Lemma *preservation_of_evaluations* :
for all $c\ s\ l\ c'\ s'\ l'$, if $c\ ^{l}\searrow_{l'}\ c'$ and $\mathtt{Some}\ s\ -\ c\ \rightarrow\ \mathtt{Some}\ s'$, then
$\mathtt{Some}\ (l, s)\ \succ\ c'\ \rightarrow\ \mathtt{Some}\ (l + \mathrm{card}(\mathrm{dom}(c')), s')$.
Lemma *preservation_hoare* :
for all $\mathcal{P}, \mathcal{Q}, c$ such that $\{\mathcal{P}\}\,c\,\{\mathcal{Q}\}$ and for all l, c', l' such that $c\ ^{l}\searrow_{l'}\ c'$ then
$[\lambda pc.\lambda s.\ pc = l \wedge \mathcal{P}\ s]\,c'\,[\lambda pc.\lambda s.\ pc = l' \wedge \mathcal{Q}\ s]$.

5 Extraction of the Semantics of BBS in Assembly

5.1 The Functional Correctness of BBS in Assembly

Let us provide two functions encode and decode such that: $\mathsf{encode}\,(n, k, seed, m)$ builds a state from the requested number n of pseudoramdom 32-bits words, the number k of 32-bits words reserved for the encoding of the seed and the modulus, the *seed*, and the modulus m; and $\mathsf{decode}\,(s)$ is the list of pseudorandom bits stored in the state s. These functions impose a specific memory layout depicted in Fig. 5. Besides the encoding of the seed (in memory area X) and the modulus (in M) as multi-precision integers, and initialization of the list of pseudorandom bits (in L) to an appropriate length, encode provides additional storage $(Y, \beta^{2k}$, trailing words initialized to $0_{32})$ specific to our implementation (this stems from our Montgomery multiplication). Note that, as long as $4(4k + n + 2) < 2^{32}$, n and k can be very large, k effectively covering lengths for which the quadratic residuosity problem is indeed believed to be intractable. This is one desirable side-effect of our approach to precisely pinpoint the range of k.

Using above functions, the verification goal is stated as follows:

$$\left[\begin{array}{c} \lambda pc.\lambda s.\ pc = 0\ \wedge \\ \mathsf{encode}\,(n, k, seed, m) = s \end{array}\right] \mathsf{bbs\_asm} \left[\begin{array}{c} 4(4k + n + 2) < 2^{32} \rightarrow \\ \lambda pc.\lambda s.\ pc = 232\ \wedge \\ \mathsf{decode}\,(s) = \mathsf{bbs\_fun}(32 \times n, seed, m) \end{array}\right]$$
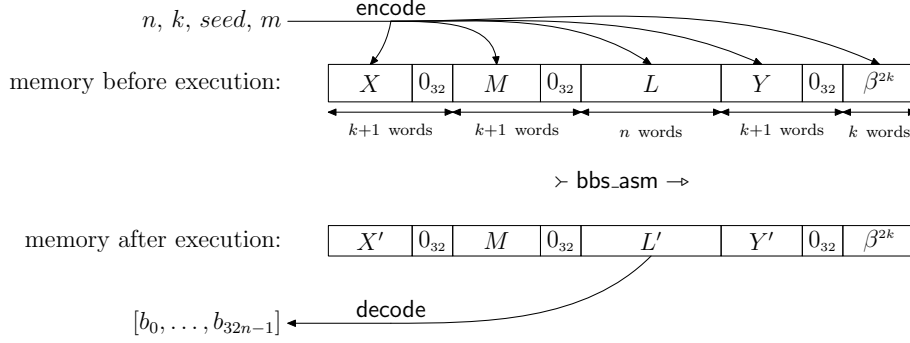
Figure 5: Encoding and decoding of input/output

Starting from an appropriate encoding of the inputs, the execution of bbs_asm leads to a final state from which one can extract the intended list of pseudorandom bits (see [Code] for details). It is here that the restrictions imposed by the choice of implementation mentioned in Sect. 3 appear, for the above triple cannot be proved for arbitrary values of n and k.

In practice, we conduct formal proof using standard Separation Logic and obtain the triple above by applying the lemma *preservation_hoare* of Sect. 4.3. The effort therefore concentrates on the following triple where the assembly program with jumps has been replaced by its "decompiled" version with if-then-else's and while-loops (manual decompilation proved correct w.r.t. the compilation predicate of Sect. 4.3):

$$4(4k + n + 2) < 2^{32} \rightarrow$$
$$\{\lambda s. \, \mathsf{encode}\,(n, k, seed, m) = s\} \, \mathsf{bbs\_asm\_decompile} \, \{\lambda s. \, \mathsf{decode}\,(s) = \mathsf{bbs\_fun}(32{\times}n, seed, m)\} \tag{1}$$

Note that we are dealing with a generalized version of the BBS algorithm (bbs_fun takes the modulus m in $\mathbb{Z}$, whereas bbs in Sect. 2.1 uses the types $\mathbb{Z}_m^*$ and QR_m):

$$\mathsf{bbs\_fun}(len \in \mathbb{N}, seed \in \mathbb{Z}, m \in \mathbb{Z}) =_{def} \mathsf{bbs\_fun\_rec}(len, seed^2 \pmod m, m)$$
$$\mathsf{bbs\_fun\_rec}(len \in \mathbb{N}, x \in \mathbb{Z}, m \in \mathbb{Z}) =_{def}$$
$$\mathbf{match} \; len \; \mathbf{with} \; 0 \; \Rightarrow \; [] \; | \; len' + 1 \; \Rightarrow \; \mathsf{parity}(x) :: \mathsf{bbs\_fun\_rec}(len', x^2 \pmod m, m) \; \mathbf{end}$$

This is a sound generalization because the information that $\mathbb{Z}_m^*$ is a cyclic group is not needed in the proof of functional correctness (only in the cryptographic proof).

5.2   Extraction of the Semantics of BBS in Assembly

First, we prove that bbs_asm is terminating and deterministic, i.e., for all n, k, $seed$ and m, there exists a unique state s' such that $\mathsf{Some}\,(0, \mathsf{encode}\,(n, k, seed, m)) \succ \mathsf{bbs\_asm} \rightarrow s'$. From the Separation Logic triple of the previous section, we derive by soundness of Separation Logic:

Lemma *correctness* :
if $\mathsf{Some}\,(0, \mathsf{encode}\,(n, k, seed, m)) \succ \mathsf{bbs\_asm} \rightarrow \mathsf{Some}\,(l', s')$ and $4(4k + n + 2) < 2^{32}$,
then $l' = 232$ and $\mathsf{decode}\,(s') = \mathsf{bbs\_fun}(32{\times}n, seed, m)$

Then comes the proof of termination. First, we prove that there is a final state, without proving whether it is an error state or not:

Lemma *execution_bbs_asm* :
if $4(4k + n + 2) < 2^{32}$, then there exists s' s. t. $\mathsf{Some}\,(0, \mathsf{encode}\,(n, k, seed, m)) \succ \mathsf{bbs\_asm} \rightarrow s'$

This is proved by induction on the variant of the outermost loop, and then on nested loops. Second, by the triple (1), we derive the fact that this final state cannot be an error state.

9

The lemmas above allow us to define a function $\mathsf{exec}_{\mathsf{bbs\_asm}}$ that maps a number n of 32 bits words, a number k of 32 bits words, a *seed* and a modulus m to a state from which one can extract the desired pseudorandom bits, so that the semantics of $\mathsf{bbs\_asm}$ can be written as a mathematical function:

$$[\![\mathsf{bbs\_asm}]\!] =_{def} \mathsf{prefix}_{len+1} \left(\mathsf{decode} \left(\mathsf{exec}_{\mathsf{bbs\_asm}} \left(\left\lceil \frac{len+1}{32} \right\rceil, \lceil \log_{2^{32}}(m) \rceil, seed, m \right) \right) \right)$$

Since $\mathsf{bbs\_asm}$ always return a number of pseudorandom bits that is a multiple of 32, we need to take a prefix of its output; $len + 1$ is the length requested by the unpredictability game.

Putting it All Together The Figure below summarizes how we organize the complete game-based proof of BBS in assembly. `bbs_encode_decode.v` contains the `encode`/`decode` functions and the formal proof of correctness and termination, derived from the formal proof of the Separation Logic triple for $\mathsf{bbs\_asm}$. `BBS_Asm_CryptoProof.v` contains the game-based proof, making use of the correctness and termination lemmas and of the cryptographic proof of the BBS algorithm provided by `BBS.v`, taken directly from [Now08].
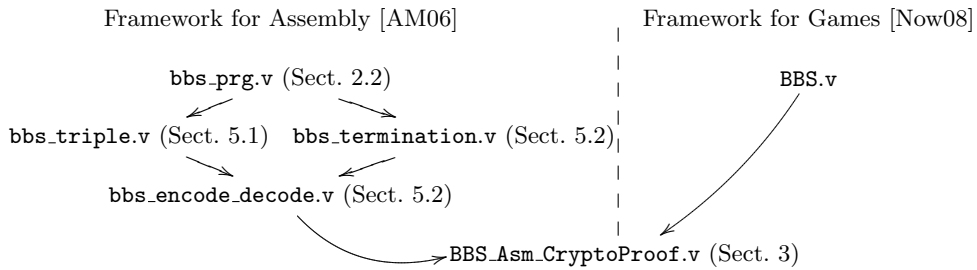
6 Technical Aspects of the Coq Formalization

The formalization of assembly programs, operational semantics, Separation Logic, as well as all supporting lemmas is the result of a revision of our previous work [AM06]. This revision was made necessary to address scalability issues. We do not comment extensively about this revision except to say that we used SSREFLECT [GM07], a recently publicized Coq extension, that favors a proof style that naturally led to shorter proof scripts (for illustration, proof scripts of experiments in [AM06] shrank by 70% in terms of lines of code).

The new aspect of our framework is the formalization of the proof-carrying code framework of [SU07], that we instantiate to Separation Logic and MIPS instructions and extend to deal with error-states. Table 1 makes it clear what is formalized w.r.t. [SU07]. In brief, what we do not do: we do not formalize Section 5 of [SU07] and we formalize only the so-called "non-constructive proofs" of Theorems 17 and 18 (indeed, for these two theorems, the proofs come in two flavors).

The formal proof of the Separation Logic triple of Sect. 5.1 is technically the most demanding part of the proof effort. Our assembly program of BBS is large (at least by the current standards of proof assistant-based verification [AM06, MG07]): 239 instructions that spread over several snippets of code. Table 6 makes it precise which snippets are used and their respective size.

Table 3 summarizes the size of proof scripts used in the proof of the Separation Logic triple of BBS. It is always difficult to comment about the size of proof scripts because we are lacking good metrics for comparison. Yet, looking at related work, it is fair to claim that our framework for formal proof of assembly programs allows for short proof scripts: this can be appreciated by looking at several similar experiments in common among the work in this paper and [AM06, MG07] (verification of multi-precision arithmetic, Montgomery multiplication, but also Montgomery exponentiation, not used in this paper though).

Reference in [SU07]	Reference in [Code] and status	Proof script size
Section 2	file `goto.v`	462 lines
Figure 1, Lemma 1, 3	Done	
Lemma 2	Particular cases only	
Section 3	file `sgoto.v`	747 lines
Section 3.1: Figure 2, Lemmas 4–5, Theorems 6–8, Corollary 9	Done	
Section 3.2: Figure 3, Theorem 10, Lemma 11, Theorem 12	Done	
Section 4	file `compile.v`	1279 lines
Section 4.1: Figure 5, Lemmas 13–14, Theorems 15–16	Done	
Section 4.2: Theorems 17–18	Done	
Section 4.3	Done, file `sgoto_hoare.v`	369 lines
Section 5	Not done	
Appendix A	Done (revision of [AM06])	
	file `mips_biple.v`	927 lines
	file `mips_cmd.v`	710 lines
	file `mips_hoare.v`	783 lines
Appendix B		
Theorems 6–7, 15–18	Done (spread over above files)	

Table 1: Formalization of [SU07]

Function	Reference in [Code]	Program size
BBS	`bbs_prg.v`	14 commands
Montgomery strict (Fig. 2.3)	`mont_mul_strict_prg.v`	9 commands
Montgomery raw ([AM06])	`mont_mul_prg.v`	36 commands
Multi-precision subtraction	`multi_sub_prg.v`	18 commands
Multi-precision comparison	`multi_lt_prg.v`	11 commands
Array initialization	`multi_zero_prg.v`	6 commands

Table 2: The assembly code of BBS in Coq

7   Conclusion

We addressed the problem of formal verification of assembly code with cryptographic proofs. We proposed an approach that extends game-based proofs to integrate formal proofs of functional correctness with formal cryptographic proofs in a clear way, understandable by both cryptographers and implementers. Our proposition is supported by a concrete framework developed in the Coq proof assistant. As an illustration, we provided the first assembly program for a pseudorandom number generator that is certified with a cryptographic proof.

Future Work The cryptographic proof of BBS on which we rely is asymptotic: the probability that an attacker predicts the next bit can be made arbitrarily small, but it does not give any concrete value for the security parameter. A possible extension of our approach would be to link our assembly implementation of BBS to a cryptographic proof of the *concrete security* of BBS.

Our certified implementation of BBS could be used as the source of pseudorandomness in the implementation of further cryptographic primitives. Indeed, even though it is probabilistic, such a primitive is still deterministic in the sense that for any two equal inputs it outputs the same distribution; one can thus extract its semantics as a mathematical function and inject it into the appropriate standard definition of security (such as *semantic security* in the case of ElGamal).

Function	Reference in [Code]	Size
BBS	`bbs_triple.v`	845 lines
Montgomery strict	`mont_{mul,square}_strict_init_triple.v`	608 lines
Montgomery raw	`mont_{mul,square}_triple.v`	1198 lines
Multi-precision subtraction	`multi_sub_inplace_left_triple.v`	439 lines
Multi-precision comparison	`multi_lt_triple.v`	408 lines
Array initialization	`multi_zero_triple.v`	129 lines
Total		3627 lines

Table 3: Formal proof of the Separation Logic triple of BBS

References

[ATM07] Affeldt, R., Tanaka, M., Marti, N.: Formal Proof of Provable Security by Game-Playing in a Proof Assistant. Int. Conf. on Provable Security. LNCS, vol. 4784, pp. 151–168. Springer (2007).

[AM06] Affeldt, R., Marti, N.: An Approach to Formal Verification of Arithmetic Functions in Assembly. Annual Asian Computing Science Conference, Dec. 2006. LNCS, vol. 4435, pp. 346–360. Springer, Heidelberg (2008).

[BBU08] Backes, M., Berg, M., Unruh D.: A Formal Language for Cryptographic Pseudocode. Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning. LNCS, vol. 5330, , pp. 353–376. Springer (2008).

[BGZ09] Barthe, G., Grégoire, B., Zanella, S.B.: Formal certification of code-based cryptographic proofs. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp.90–101. ACM Press.

[BR04] Bellare, M., Rogaway, P.: Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004.

[BBS86] Blum, L., Blum, M., Shub, M.: A simple unpredictable pseudo random number generator. SIAM Journal on Computing, 15(2):364–383. Society for Industrial and Applied Mathematics, 1986.

[Code] Affeldt, R., Nowak D., Yamada K.: Certifying Assembly with Cryptographic Proofs: the Case of BBS. http://staff.aist.go.jp/reynald.affeldt/bbs

[GM07] Gonthier, G., Mahboubi, A.: A Small Scale Reflection Extension for the Coq System. Technical Report 6455, Dec. 2007. INRIA.

[Mips] MIPS Technologies. MIPS32 4KS Processor Core Family Software User's Manual MIPS Technologies, Inc., 1225 Charleston Road, Mountain View, CA 94043-1353.

[Mon85] Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation, 44(170):519–521, 1985.

[MG07] Myreen, M.O., Gordon, M.J.C.: Verification of Machine Code Implementations of Arithmetic Functions for Cryptography. Theorem Proving in Higher Order Logics: Emerging Trends Proceedings. Internal Report 364/07, Aug. 2007. Department of Computer Science, University of Kaiserslautern.

[MSG09]   Myreen, M.O., Slind, K., Gordon, M.J.C.: Extensible proof-producing compilation. Int. Conf. on Compiler Construction. LNCS, vol. 5501, pp. 2–16. Springer (2009).

[Now07]   Nowak, D.: A framework for game-based security proofs. Int. Conf. on Information and Communications Security. LNCS, vol. 4861, pp. 319–333. Springer (2007).

[Now08]   Nowak, D.: On formal verification of arithmetic-based cryptographic primitives. Int. Conf. on Information Security and Cryptology, Dec. 2008. LNCS, vol. 5461, pp. 368-382. Springer (2009).

[Rey02]   Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. IEEE Symp. on Logic in Computer Science, pp. 55–74 (2002). Invited lecture.

[SU07]   Saabas, A., Uustalu, T.: A compositional natural semantics and Hoare logic for low-level languages. Theoretical Computer Science 373(3), 273–302. Elsevier (2007).

[Sho04]   Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.

[Yao82]   Yao, A.C.: Theory and applications of trapdoor functions. IEEE Annual Symp. on Foundations of Computer Science. pp. 80–91. IEEE (1982).