Certifying Assembly with Formal Cryptographic Proofs: the Case of BBS

Reynald Affeldt, David Nowak and Kiyoshi Yamada
Research Center for Information Security, AIST, Japan

July 1, 2009

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 extend security proofs to guarantee the security of assembly 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

Embedded cryptographic software is now a crucial part of security in our modern digital societies. As a consequence, public institutions as well as private manufacturers recognize, through international standards for computer security such as the Common Criteria (ISO/IEC 15408), a pressing need for *certification*. Given its central role, embedded cryptographic software deserves the most trustful level of certification: formal verification using computer-checkable formal proofs, as performed by proof-assistants based on proof-theory. However, as of today, despite the fact that most security claims implicitly assume correct implementation of cryptography, this assumption is almost never formally enforced in practice. With more and more processing of sensitive data delegated everyday to embedded systems such as smartcards, it becomes important to provide practical solutions for certification of embedded cryptographic software.

The main problem of certification of embedded cryptographic software is that, in the current state of research, certification 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. 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, certification of embedded cryptographic software is even more challenging 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.

This paper addresses the issue of certifying assembly code with cryptographic proofs. As pointed above, certification of assembly code and certification 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 already exist ([AM06, MG07] for cryptographic assembly code, [ATM07, Now07, BGZ09, BBU08] 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 clear way, both understandable by cryptographers and implementers, and in a reusable fashion, so that new certification efforts can build upon previous ones.

Contributions In this paper, our main contribution is to propose a concrete approach, supported by a reusable formal framework, for certifying assembly code together with cryptographic proofs. As a concrete evidence of usability, we certify a pseudorandom number generator written in assembly for smartcards with a formal proof of unpredictability. This choice of application is not gratuitous: this is the natural first step before certifying more cryptographic primitives, since many of them actually rely on pseudorandom number generation.

To achieve our goal, we integrate two existing formal 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]. Game-playing is a popular setting to represent cryptographic proofs: security definitions and computational assumptions are represented as *games* between which one performs incremental *game transformations*. We introduce a new kind of game transformation, that we call *implementation step*, 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.
- To support the above integration, we extend the formal framework for assembly code of [AM06] to connect with the formal framework for cryptographic proofs of [Now07] (both in the Coq proof-assistant). 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. In this paper, we explain in particular how we deal with arbitrary jumps in assembly. Concretely, we provide an original formalization of the proof-carrying code framework of [SU07], that allows us to certify assembly with jumps through standard Hoare logics proofs.
- We provide the first snippet of assembly code for a pseudorandom number generator that is certified 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 certification of hand-written assembly. For this purpose, the extension of [AM06] that we propose in this paper is well-suited because it provides already enough material for reasonably short proof scripts. Under less constraints about the target code, the alternative approach of proof-producing compilation [MSG09] could be considered. Of course, a compiler such as the one of [MSG09] needs to be extended with custom support for cryptography to achieve reasonable performance. Still, an early application of ideas of proof-producing compilation to cryptographic functions shows that short proof scripts and compact assembly are difficult to reconcile [MG07]. Overcoming these difficulties is still not enough for our overall goal, for HOL (the proof assistant used in [MSG09]) lacking a framework for formal cryptographic proofs.

The two previous work ([AM06] and [Now07]) that we integrate in this paper turn out to be a good fit for they favoring 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 [BGZ09, BBU08] 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 This paper is organized as follows. In Sect. 2, we introduce the BBS algorithm and provide an assembly implementation. In Sect. 3, we explain how we integrate formal proofs of functional correctness for assembly code with game-based formal 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 more specifically the formal proof of functional correctness of BBS and the lemmas relevant to the integration with its cryptographic proof. In Sect. 6, we wrap up the complete formal cryptographic proof of BBS in assembly by giving an overview 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 [MOV96]. In fact, 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. The BBS algorithm is written below as a Coq function. It performs iteratively squaring modulo and outputs the result of parity tests. The input consists of 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.

$$\begin{array}{l}\mathsf{bbs}(len \in \mathbb{N}, seed \in \mathbb{Z}_m^*) \overset{def}{=} \mathsf{bbs\_rec}(len, seed^2)\\ \mathsf{bbs\_rec}(len \in \mathbb{N}, x \in QR_m) \overset{def}{=}\\ \quad \textbf{match}\ len\ \textbf{with}\ 0 \ \Rightarrow\ [] \mid len' + 1 \ \Rightarrow\ \mathsf{parity}(x) :: \mathsf{bbs\_rec}(len', x^2)\ \textbf{end}\end{array}$$

BBS is one of the rare pseudorandom number generators that is *cryptographically secure* [BBS86]. This means that it passes all polynomial-time statistical tests, i.e., 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 more precise terms, 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].) A formal account of the proof of unpredictabiliy of the BBS algorithm can be found in [Now08], where it is presented in the game-playing formalism [Sho04].[1]

2.2 Implementation of BBS in Assembly

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

In brief, it consists of a loop with a nested loop. Each iteration of the nested loop produces one word of pseudorandom bits. More precisely, the nested loop performs a square modulo, extracts the parity bit (of the least significant word), and stores 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;

[1]It is shown in [VV84] that BBS is still cryptographically secure under the weaker assumption that m is hard to factorize. The same authors also show that, for sufficiently large m, more than one bit can be extracted at each iteration of the algorithm. However, in this paper, we stick to the original proof of [BBS86].

```
bbs_asm $\overset{def}{=}$
0:     addiu $i$ gpr_zero $0_{16}$                  (* init counter for outer loop *)
1:     addiu $L$ $l$ $0_{16}$                               (* init pointer to result *)
2:     beq $i$ $n$ 240                                        (* repeat $n$ times *)
3:       addiu $j$ gpr_zero $0_{16}$                 (* init counter for inner loop *)
4:       addiu $w$ gpr_zero $0_{16}$     (* init word of temporary storage *)
5:       beq $j$ $thirtytwo$ 236                             (* repeat 32 times *)
6:         $\boxed{\text{mul\_mod } k\ x\ x\ m \ldots}$            (* compute $X^2$ (mod $M$) *)
230:       lw $w\_$ $0_{16}$ $x$                  (* load least significant word *)
231:       andi $w\_$ $w\_$ $1_{16}$                            (* extract parity bit *)
232:       sllv $w\_$ $w\_$ $j$                (* shift parity bit to $j$th position *)
233:       cmd_or $w$ $w$ $w\_$ (* store parity bit in temporary storage *)
234:       addiu $j$ $j$ $1_{16}$                (* increment inner loop counter *)
235:       jmp 5                                          (* end of the inner loop *)
236:     sw $w$ $0_{16}$ $L$       (* store the last 32 parity bits in memory *)
237:     addiu $L$ $L$ $4_{16}$                   (* increment pointer to result *)
238:     addiu $i$ $i$ $1_{16}$                (* increment outer loop counter *)
239:     jmp 2                                            (* end of the outer loop *)
240:
```

Figure 1: The assembly code bbs_asm

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 stands for an inlined snippet of assembly code whose purpose is to compute the multi-precision square modulo (see Appendix E for details).

3 Game-based Proofs for Assembly

Cryptographic proofs usually apply to algorithms without any consideration for implementation aspects. In this section, we propose a way to lift a standard definition of unpredictability (the security notion of interest when dealing with pseudorandom number generation) so as to formally prove unpredictability directly on assembly code.

We choose to work in the setting of game-playing [Sho04] because it lends itself easily to formalization and extension. Game-playing is a methodology for writing cryptographic proofs that are easier to verify. In game-playing, a security property is modeled as a program that implements a game to be solved by the attacker, and the attacker is modeled as an external probabilistic procedure interfaced with the game. A cryptographic proof consists in showing that any attacker has only little advantage over a random player. This is achieved by applying the security definition to the cryptographic primitive to be verified, and then by reducing it to a computational assumption through game transformations. Game-playing lends itself easily to formalization in proof-assistants [BBU08, ATM07, Now07, BGZ09] and it also lends itself easily to extension, since reasoning steps are identified with game transformations. Indeed, the lifting we propose can be simply thought of as a new kind of game transformation.

3.1 The Unpredictability Game

The unpredictability game is defined as follows: 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 the function f; this sequence, deprived of its first bit b_0, is passed to the probabilistic attacker A; the latter returns its guess $\widehat{b_0}$ for the value of this bit. The result of the game is whether the guess is right or not. Using the notations of [Now08], this game is written:

$$\mathsf{unpredictability}(f) \overset{def}{=} \left(\begin{array}{l} seed \overset{R}{\leftarrow} G \ ; \\ [b_0, \ldots, b_{len}] \leftarrow f(len+1, seed) \ ; \\ \widehat{b_0} \Leftarrow A([b_1, \ldots, b_{len}]) \ ; \\ \textbf{return} \ \widehat{b_0} \overset{?}{=} b_0 \end{array} \right)$$

3.2 The Unpredictability Game for Assembly

We propose to define the unpredictability game directly on the assembly code by lifting the above standard definition. Lifting is needed because the standard definition applies to mathematical functions without any consideration for their concrete implementation. This makes a difference because, contrary to mathematical functions, assembly code does not work as intended for any input due to restrictions imposed by the choice of implementation.

The basic idea for the lifting is to extract from the assembly code its semantics in terms of a mathematical function and to inject it into the standard definition of the unpredictability game. For such a function to exist, the assembly code c has to be terminating and deterministic. More importantly, one also needs to make clear under which restrictions the assembly code behaves as intended. Under above conditions, one can define a total function $[\![c]\!]$ on which we can apply the standard definition of unpredictability game. This leads to the following definition of unpredictability game for assembly code:

$$\mathsf{unpredictability\_assembly}(c) \overset{def}{=} \left(\begin{array}{l} seed \overset{R}{\leftarrow} G \ ; \\ [b_0, \ldots, b_{len}] \leftarrow [\![c]\!](len+1, seed) \ ; \\ \widehat{b_0} \Leftarrow A([b_1, \ldots, b_{len}]) \ ; \\ \textbf{return} \ \widehat{b_0} \overset{?}{=} b_0 \end{array} \right)$$

3.3 Implementation Steps as Game Transformations

The advantage of the lifting explained above is that it makes it clear how to organize certification of assembly code with cryptographic proofs: games for assembly connect to standard games through *implementation steps*. An implementation step can be justified formally by proving that the assembly code does implement the intended cryptographic primitive. And this cannot be achieved without making explicit the restrictions imposed by the choice of implementation, as intended.

Since implementation steps come in addition to the other three types of game transformations identified by Shoup [Sho04] (indistinguishability steps, failure event steps, and bridging steps), this makes it easier to develop a formal framework for certification of assembly with cryptographic proofs: pick up a formal framework for game-based proofs and a formal framework to verify assembly, and add the machinery to perform implementation steps. The next section illustrates this approach in the case of BBS, the rest of this paper explains how we integrate [AM06] and [Now07] for this purpose.

3.4 Application to BBS

In this section, we show how to add an implementation step at the beginning of the cryptographic proof of [BBS86] for the BBS algorithm so as to get a cryptographic proof for the BBS assembly code. At this stage, we only give an overview; detailed explanations are given in the next sections of this paper.

Let us assume that we are given an operational semantics for assembly code as a predicate written $s \succ c \twoheadrightarrow s'$, meaning that from state s, the execution of the assembly code c terminates in the state s'. A state is of the form either `Some` (l, s) where l is the program counter and s is a memory, or `None` for error states (see Sect. 4.3 for details).

Let us assume that we are given 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 *seed*, and the modulus m; and $\mathsf{decode}\,(s)$ is the list of pseudorandom bits stored in the state s (see Sect. 5.1 for details). We first 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 $\texttt{Some}\ (\mathsf{encode}\,(n,k,seed,m)) \succ \mathsf{bbs\_asm} \twoheadrightarrow s'$ (for the sake of simplicity, we assume here that the encode/decode functions also take care of labels, see Sect. 5.2 for details). This gives a function $\mathsf{exec}_{\mathsf{bbs\_asm}}$ that maps any state s to the state s' such that $\texttt{Some}\ s \succ \mathsf{bbs\_asm} \twoheadrightarrow s'$. This allows us to define $[\![\mathsf{bbs\_asm}]\!]$ which maps a length $len+1$, a seed $seed$ and a modulus m to the list of bits:

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

Game 1 We start with the unpredictability game for bbs_asm:

$$\left(\begin{array}{l} seed \overset{R}{\leftarrow} \mathbb{Z}_m^* \ ; \\ [b_0,\ldots,b_{len}] \leftarrow [\![\mathsf{bbs\_asm}]\!](len+1, seed) \ ; \\ \widehat{b_0} \Leftarrow A([b_1,\ldots,b_{len}]) \ ; \\ \textbf{return } \widehat{b_0} \overset{?}{=} b_0 \end{array}\right)$$

Game 1′ The first step is to unfold $[\![\mathsf{bbs\_asm}]\!]$:

$$\left(\begin{array}{l} seed \overset{R}{\leftarrow} \mathbb{Z}_m^* \ ; \\ [b_0,\ldots,b_{len}] \leftarrow \\ \quad \mathsf{prefix}_{len+1}\left(\mathsf{decode}\left(\mathsf{exec}_{\mathsf{bbs\_asm}}\left(\mathsf{encode}\left(\left\lceil\frac{len+1}{32}\right\rceil, \lceil\log_{2^{32}}(m)\rceil, seed, m\right)\right)\right)\right) \ ; \\ \widehat{b_0} \Leftarrow A([b_1,\ldots,b_{len}]) \ ; \\ \textbf{return } \widehat{b_0} \overset{?}{=} b_0 \end{array}\right)$$

Game 2 Next comes the implementation step that connects the above game to the initial game of the cryptographic proof of the BBS algorithm [Now08]. For this purpose, we first prove that bbs_asm is correct, i.e., for all $n, k, seed, m, s'$, such that $\texttt{Some}\ (\mathsf{encode}\,(n,k,seed,m)) \succ \mathsf{bbs\_asm} \twoheadrightarrow \texttt{Some}\ s'$, then $\mathsf{decode}\,(s') = \mathsf{bbs}(32 \times n, seed)$ where bbs is the BBS algorithm of Sect. 2.1 (see Sect. 5 for details). This gives rise to the following game:

$$\left(\begin{array}{l} seed \overset{R}{\leftarrow} \mathbb{Z}_m^* \ ; \\ [b_0,\ldots,b_{len}] \leftarrow \mathsf{bbs}(len+1, seed) \ ; \\ \widehat{b_0} \Leftarrow A([b_1,\ldots,b_{len}]) \ ; \\ \textbf{return } \widehat{b_0} \overset{?}{=} b_0 \end{array}\right)$$

This is actually the first game of the cryptographic proof of unpredictability of the BBS algorithm as formalized in [Now08]. That concludes the cryptographic proof of unpredictability of bbs_asm.

4 Verification of Functional Correctness of Assembly

In the previous section, we saw that, in order to perform cryptographic proofs of an implementation, we need in particular to prove its functional correctness. This is in particular technically difficult for assembly. Indeed, handling of jumps results in non-standard logics, less practical than standard Hoare logic (see [SU07] for references). To overcome this difficulty, we propose to formalize the proof-carrying code framework of [SU07]. This framework provides a compositional operational semantics and Hoare logic for assembly with jumps. However, derivations in this operational semantics and this Hoare logic have the shortcoming of being verbose. Fortunately, [SU07] also shows that these derivations can be obtained from standard operational semantics and standard Hoare logic by compilation from assembly code with while-loops to assembly code with jumps. In other words, compilation let us work with standard Hoare logic (with while-loops) while still being able to recover formal proofs for assembly with jumps.

In this section, we comment on our formalization of [SU07] and its instantiation to Separation Logic and MIPS.

4.1 Formalization of States

A state consists of a *store* and a *heap*. The store is a finite map from registers to 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. The heap is a finite map from locations to integers of type int_{32}. The heap is tailored to word-accesses for convenience 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.

We found it convenient in practice to separate general-purpose registers from the *multiplier* (the subset of registers dedicated to arithmetic computations), hence the following definition of states:

$$state \overset{def}{=} store \times multiplier \times heap$$

Finally, 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:

$$lstate \overset{def}{=} \texttt{option}\ (label \times state)$$

4.2 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. Formalizing 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, $(v)_{32 \to \mathbb{N}}$ interprets the value v as an unsigned integer, etc.

We illustrate the semantics of MIPS instructions with the rules for the instruction `lw` ("load word"). The formal semantics of other instructions (29 in total) can be found in [Code], file `mips.v`. For `lw`, there are two rules depending on whether the memory access is word-aligned and the accessed location is undefined. (The notation $+_h$ below is for the addition on finite-size integers.)

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

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

4.3 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 branching MIPS instructions are parameterized by conditions noted b below. Given an assembly program c, dom(c) is the set of the labels of the instructions of c. Labelled instructions are assembled using a composition operator $\oplus$ (this notation stems from the fact that sets of labelled instructions exhibit a structure of commutative monoid—see [SU07] for explanations and [Code] for formal proofs).

The operational semantics of assembly programs is a predicate noted $s\ \succ\ c\ \multimap\ s'$ 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 below. 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 can be thought as a mix of the rules for sequence and while-loops of traditional Hoare logic.

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

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

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

$$\frac{\begin{array}{c} l \in \mathrm{dom}(c_1) \\ \texttt{Some}\ (l,s) \succ c_1 \rightarrowtriangle s' \quad s' \succ c_1 \oplus c_2 \rightarrowtriangle s'' \end{array}}{\texttt{Some}\ (l,s)\ \succ\ c_1 \oplus c_2\ \rightarrowtriangle\ s''} \qquad \frac{\begin{array}{c} l \in \mathrm{dom}(c_2) \\ \texttt{Some}\ (l,s) \succ c_2 \rightarrowtriangle s' \quad s' \succ c_1 \oplus c_2 \rightarrowtriangle s'' \end{array}}{\texttt{Some}\ (l,s)\ \succ\ c_1 \oplus c_2\ \rightarrowtriangle\ s''}$$

$$\frac{}{\texttt{None}\ \succ\ c\ \rightarrowtriangle\ \texttt{None}} \qquad \frac{l \notin \mathrm{dom}(c)}{\texttt{Some}\ (l,s)\ \succ\ c\ \rightarrowtriangle\ \texttt{Some}\ (l,s)}$$

4.4 Compilation from Standard Semantics and Hoare Logic

The non-standard operational semantics of the previous section also gives rise to a non-standard Hoare logic for assemblies with jumps. [SU07] shows that derivations for this non-standard operational semantics and Hoare logic can actually be obtained from standard operational semantics and Hoare logic through compilation. This is a result of interest because it allows 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).

In this section, we give an overview of the Hoare logics (actually, extensions known as Separation Logic [Rey02]) that we have formalized and the lemmas from [SU07] whose formal versions are involved in our verification of BBS.

Assertions Properties of states are specified using a shallow-encoding of logical connectives. This means that assertions in Separation Logic are functions from states to the type **Prop** of propositions in Coq (hereafter, assertions are displayed in calligraphic style to ease reading):

$$\mathit{assertion} \overset{def}{=} \mathit{store} \rightarrow \mathit{multiplier} \rightarrow \mathit{heap} \rightarrow \mathbf{Prop}$$

The satisfiability of an assertion can depend on the value of the current label:

$$\mathit{assn} \overset{def}{=} \mathit{label} \rightarrow \mathit{assertion}$$

The encoding being shallow facilitates the encoding of connectives (this is illustrated in Appendix A.1).

Hoare Logics We formalized two Separation Logics:

- A 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. This logic is formally proved sound and complete w.r.t. the big-step operational semantics of Sect. 4.3. (See Appendix B for the complete formalization of this Separation Logic.)
- The 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. This logic is formally proved sound and complete w.r.t. standard big-step operational semantics (see Appendix A.2 and Appendix A.3) for a reminder).

4.4.1 Compilation

[SU07] provides a compilation procedure that 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 (see Appendix C for the complete predicate). Through compilation, derivations of operational semantics can be compiled from the standard semantics (see Appendix A.2) to the non-standard semantics of Sect. 4.3:

Lemma *preservation_of_evaluations* :
for all $c\ s\ l\ c'\ s'\ l'$, if $c\ {}^{l}\!\searrow_{l'}\ c'$ and $\texttt{Some}\ s\ -\ c\ \multimap\ \texttt{Some}\ s'$, then
$\texttt{Some}\ (l, s)\ \succ\ c'\ \multimap\ \texttt{Some}\ (l + \mathrm{card}(\mathrm{dom}(c')), s')$.

Through compilation, proofs in Separation Logic can be compiled from the standard logic to the non-standard logic of [SU07]:

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
$[\lambda pc.\lambda s.\ pc = l \wedge \mathcal{P}\ s]\ c'\ [\lambda pc.\lambda s.\ pc = l' \wedge \mathcal{Q}\ s]$.

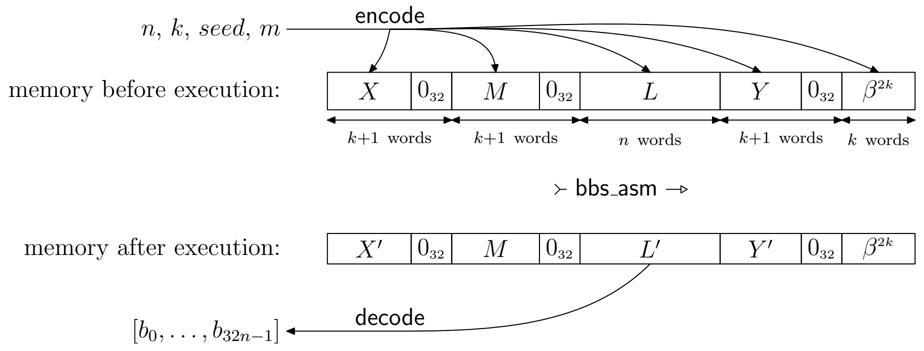
Using these results, one can carry out formal verification of a program with while-loops using standard Hoare logic and afterwards compile to obtain a formal proof for the corresponding assembly program (with jumps). In Appendix D.1, we comment on the complete formalization of [SU07] in the Coq proof-assistant.

5 The Functional Correctness of BBS in Assembly

In Sect. 3, we saw that we need to formally prove termination and correctness of our implementation of Sect. 2.2 (w.r.t. to the operational semantics given in Sect. 4.3) to complete the cryptographic proof of BBS in assembly. This is the goal of this section. We start by proving functional correctness in terms of Separation Logic, from which we derive the necessary lemmas.

5.1 The Separation Logic Triple

First, we provide concrete instances for the functions encode and decode of Sect. 3. encode imposes a specific memory layout for the input of the assembly program of Sect. 2.2, as depicted below:



Besides the encoding of the seed (in memory area X) and the modulus (in memory area M) as multi-precision integers, and initialization of for the list of pseudorandom bits (memory area L) to an appropriate length, encode also provides additional storage (Y, β^{2k}, trailing words initialized to 0_{32}) that are specific to our implementation (these requirements stem for our use of the Montgomery multiplication, that calls for appropriate data structures and initialization, see Appendix E for details). 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-effects of our approach to precisely pinpoint the range of k. Using above encode and decode functions, the verification goal is stated as follows[2]:

$$\begin{gathered}4(4k+n+2) < 2^{32} \rightarrow \\ [\lambda pc.\lambda s.\ pc = 0 \wedge \mathsf{encode}\,(n, k, \mathit{seed}, m) = s] \\ \mathsf{bbs\_asm} \\ [\lambda pc.\lambda s.\ pc = 240 \wedge \mathsf{decode}\,(s) = \mathsf{bbs\_fun}(32 \times n, \mathit{seed}, m)]\end{gathered}$$

This triple states that starting from an initial state that is an appropriate encoding of the inputs, the execution of bbs_asm leads to some final state from which one can extract the intended list of pseudorandom bits. This is at this level that the restrictions imposed by the choice of implementation that we mentioned in Sect. 3 appear, for the above triple cannot be proved for any value of n and k.

Note that we are here dealing with a generalized version of the BBS algorithm that is not restricted to seeds in $\mathbb{Z}_m^*$ (bbs_fun, that explicitly takes the modulus m, in lieu of bbs—Sect. 2.1—that uses the types $\mathbb{Z}_m^*$ and QR_m). This is a sound generalization at this level because the information that $\mathbb{Z}_m^*$ is a cyclic group is only needed in the cryptographic proof, not in the proof of functional correctness.

$$\begin{aligned}&\mathsf{bbs\_fun}(\mathit{len} \in \mathbb{N}, \mathit{seed} \in \mathbb{Z}, m \in \mathbb{Z}) \overset{def}{=} \mathsf{bbs\_fun\_rec}(\mathit{len}, \mathit{seed}^2 \pmod m, m) \\ &\mathsf{bbs\_fun\_rec}(\mathit{len} \in \mathbb{N}, x \in \mathbb{Z}, m \in \mathbb{Z}) \overset{def}{=} \\ &\quad \textbf{match}\ \mathit{len}\ \textbf{with}\ 0 \Rightarrow [] \mid \mathit{len}' + 1 \Rightarrow \mathsf{parity}(x) :: \mathsf{bbs\_fun\_rec}(\mathit{len}', x^2 \pmod m, m)\ \textbf{end}\end{aligned}$$

In practice, we conduct formal proof using standard Separation Logic and obtain the triple above by applying the lemma *preservation_hoare* of Sect. 4.4.1. The effort therefore concentrates on the formal proof of the following triple where the assembly program with jumps has been replaced by its "decompiled" version (with if-then-else's and while-loops):

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

5.2 Deriving Correctness and Termination

We can now derive the lemmas of correctness and termination that are necessary to complete the game-based proof of Sect. 3. From the Separation Logic triple of the previous section, we derive by soundness of the Separation Logic:

Lemma *correctness* :
if $\mathtt{Some}\ (0, \mathsf{encode}\,(n, k, \mathit{seed}, m)) \succ \mathsf{bbs\_asm} \twoheadrightarrow \mathtt{Some}\ (l', s')$ and $4(4k+n+2) < 2^{32}$,
then $l' = 240$ and $\mathsf{decode}\,(s') = \mathsf{bbs\_fun}(32 \times n, \mathit{seed}, m)$

The proof of termination has two steps. 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' such that $\mathtt{Some}\ (0, \mathsf{encode}\,(n, k, \mathit{seed}, m)) \succ \mathsf{bbs\_asm} \twoheadrightarrow s'$

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.

[2]For the sake of clarity, we display a simplified statement; the formal, complete statement can be found in [Code]

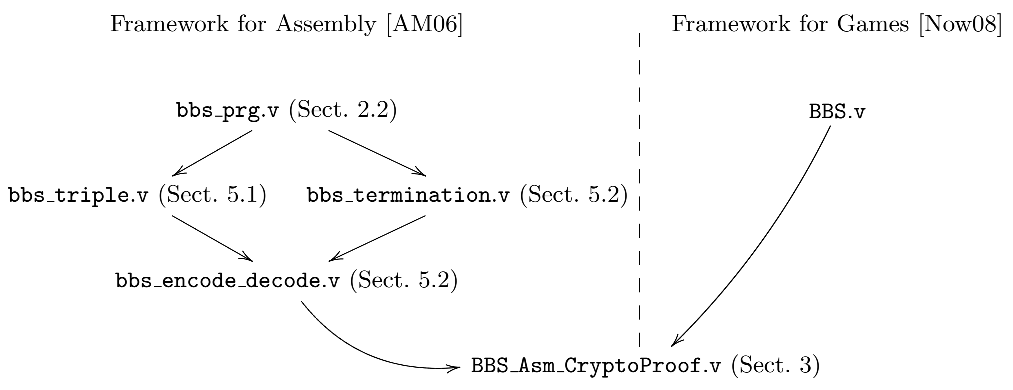

Framework for Assembly [AM06]
Framework for Games [Now08]
bbs_prg.v (Sect. 2.2)
BBS.v
bbs_triple.v (Sect. 5.1)
bbs_termination.v (Sect. 5.2)
bbs_encode_decode.v (Sect. 5.2)
BBS_Asm_CryptoProof.v (Sect. 3)

Figure 2: Summary of the formalization

6 Put it All Together

We finally explain how we organize in the Coq proof-assistant the game-based proof of BBS in assembly overviewed in Sect. 3. The formalization is summarized in Fig. 6. `bbs_encode_decode.v` essentially contains the `encode`/`decode` functions and the formal proof of correctness and termination, derived from the formal proof of the Separation Logic triple for `bbs_asm` (as explained in Sections 5.1 and 5.2). `BBS_Asm_CryptoProof.v` contains the game-based proof of Sect. 3. It makes use of the correctness and termination lemmas provided by `bbs_encode_decode.v` and of the cryptographic proof of the BBS algorithm provided by `BBS.v`, taken directly from [Now08]. See Appendix D for more insights about the technical aspects of the formalization and [Code] for the complete formalization.

7 Conclusion

In this paper, we addressed the problem of certification of assembly code with cryptographic proofs. We proposed an approach that extends game-based proofs so as 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 snippet of assembly code for a pseudorandom number generator that is certified with a cryptographic proof.

Future Work The cryptographic proof of BBS on which we rely in this paper is asymptotic. It shows that 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. One possible extension of our approach would be to link our assembly implementation of BBS to a cryptographic proof of the *concrete security* of BSS (as in [SS05]).

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 the encryption primitive of ElGamal).

References

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

[ATM07] Affeldt, R., Tanaka, M., Marti, N.: Formal Proof of Provable Security by Game-Playing in a Proof Assistant. In: 1st International Conference on Provable Security (Provsec 2007). LNCS, vol. 4784, pp. 151–168. Springer (2007).

[AM06] Affeldt, R., Marti, N.: An Approach to Formal Verification of Arithmetic Functions in Assembly. In: 11th Annual Asian Computing Science Conference, Focusing on Secure Software and Related Issues (ASIAN 2006), Dec. 2006. LNCS, vol. 4435, pp. 346–360. Springer, Heidelberg (2008).

[BBU08] Backes, M., Berg, M., Unruh D.: A Formal Language for Cryptographic Pseudocode. In: 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2008). LNCS, vol. 5330, , pp. 353–376. Springer (2008).

[BGZ09] Barthe, G., Grégoire, B., Zanella, S.B.: Formal certification of code-based cryptographic proofs. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009), 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. In: SIAM Journal on Computing, 15(2):364–383. Society for Industrial and Applied Mathematics, 1986. An earlier version appeared in the proceedings of Advances in Cryptology (CRYPTO 1982).

[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.

[MOV96] Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, 1996.

[Mon85] Montgomery, P.L.: Modular multiplication without trial division. In: 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. In: 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. In: 18th International Conference on Compiler Construction (CC 2009). LNCS, vol. 5501, pp. 2–16. Springer (2009).

[Now07] Nowak, D.: A framework for game-based security proofs. In: 9th International Conference on Information and Communications Security (ICICS 2007). LNCS, vol. 4861, pp. 319–333. Springer (2007). Also available as Cryptology ePrint Archive, Report 2007/199.

[Now08] Nowak, D.: On formal verification of arithmetic-based cryptographic primitives. In: 11th International Conference on Information Security and Cryptology (ICISC 2008), Dec. 2008. LNCS, vol. 5461, pp. 368-382. Springer (2009).

[Rey02] Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: 17th IEEE Symp. on Logic in Computer Science (LICS 2002), 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.

[SS05] Sidorenko, A., Schoenmakers, B.: Concrete Security of the Blum-Blum-Shub Pseudorandom Generator. In: 10th IMA International Conference on Cryptography and Coding. LNCS, vol. 3795, pp. 355–375. Springer (2005).

[VV84] Vazirani, U.V., Vazirani, V.V.: Efficient and secure pseudo-random number generation. In: IEEE 25th Annual Symposium on Foundations of Computer Science (FOCS 1984). pp. 458–463. IEEE (1984).

[Yao82] Yao, A.C.: Theory and applications of trapdoor functions. In: IEEE 23rd Annual Symposium on Foundations of Computer Science (FOCS 1982). pp. 80–91. IEEE (1982).

A Formalization of Standard Separation Logic

A.1 Separation Logic Connectives

For illustration, let us consider the most important connective of Separation Logic: the *separating conjunction.* Intuitively, $\mathcal{P} \star \mathcal{Q}$ holds for a state (s, m, h) when the heap can be splitted into h_1 and h_2 such that $\mathcal{P}$ holds for (s, m, h_1) and $\mathcal{Q}$ holds for (s, m, h_2). Formally:

$$\begin{array}{lcl} \mathcal{P} \star \mathcal{Q} & \overset{def}{=} & \lambda s\ m\ h.\ \exists h_1, h_2.\ \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset\ \wedge \\ & & h = h_1 \cup h_2\ \wedge\ \mathcal{P}\ s\ m\ h_1\ \wedge\ \mathcal{Q}\ s\ m\ h_2 \end{array}$$

Other important connectives in practice are the “mapsto” connectives. Let $\emptyset$ by the empty heap and let (l, v) be the singleton heap that maps location l to value v. One defines connectives that stand respectively for singleton heaps and heap-allocated arrays as follows ($+_e$ is a function symbol for addition in a language of expressions made of registers and finite-size integers):

$$\begin{array}{lcl} e \mapsto e' & \overset{def}{=} & \lambda s\ m\ h.\ \exists p.\ (\llbracket e \rrbracket_s)_{32 \to \mathbb{N}} = 4 \times p\ \wedge\ h = (p, \llbracket e' \rrbracket_s) \\ e \Mapsto l & \overset{def}{=} & \textbf{if}\ l\ \textbf{is}\ hd :: tl\ \textbf{then}\ (e \mapsto hd) \star (e +_e 4_{32} \Mapsto tl)\ \textbf{else}\ \emptyset \end{array}$$

A.2 Standard Big-step Operational Semantics

Assembly code with while-loops is built out of MIPS instructions (Sect. 4.2), sequences (notation: $c_1 \mathtt{;} c_2$), structured branching (notation: $\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$), and while-loops (notation: $\mathtt{while}\ b\ c$). Here follow the rules of the standard big-step operational semantics:

$$\frac{}{\mathtt{None}\ -\ c\ \rightarrowtriangle\ \mathtt{None}} \qquad \frac{s\ -\ c\ \longrightarrow\ s'}{s\ -\ c\ \rightarrowtriangle\ s'} \qquad \frac{s\ -\ c_1\ \rightarrowtriangle\ s'' \quad s''\ -\ c_2\ \rightarrowtriangle\ s'}{s\ -\ c_1 \mathtt{;} c_2\ \rightarrowtriangle\ s'}$$

$$\frac{\llbracket b \rrbracket_s \qquad \mathtt{Some}\ (s, m, h)\ -\ c_1\ \rightarrowtriangle\ s'}{\mathtt{Some}\ (s, m, h)\ -\ \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \rightarrowtriangle\ s'} \qquad \frac{\neg\llbracket b \rrbracket_s \qquad \mathtt{Some}\ (s, m, h)\ -\ c_2\ \rightarrowtriangle\ s'}{\mathtt{Some}\ (s, m, h)\ -\ \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \rightarrowtriangle\ s'}$$

$$\frac{\begin{array}{c}\llbracket b \rrbracket_s \quad \mathtt{Some}\ (s, m, h)\ -\ c\ \rightarrowtriangle\ s' \\ s'\ -\ \mathtt{while}\ b\ c\ \rightarrowtriangle\ s''\end{array}}{\mathtt{Some}\ (s, m, h)\ -\ \mathtt{while}\ b\ c\ \rightarrowtriangle\ s''} \qquad \frac{\neg\llbracket b \rrbracket_s}{\mathtt{Some}\ (s, m, h)\ -\ \mathtt{while}\ b\ c\ \rightarrowtriangle\ \mathtt{Some}\ (s, m, h)}$$

A.3 Standard Separation Logic

To simplify the presentation of standard Separation Logic, 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 store" instruction:

$$\begin{array}{l}
\mathcal{WP}\ i\ \mathcal{Q} \overset{def}{=} \textbf{match}\ i\ \textbf{with} \\
\mid \texttt{lw}\ rt\ \mathit{off}\ \mathit{base} \Rightarrow \lambda s.\exists p.\left(\llbracket base \rrbracket_s +_h (\llbracket \mathit{off} \rrbracket_s)_{int_{16}\to int_{32}}\right)_{32\to\mathbb{N}} = 4 \times p\ \land \\
\qquad\qquad\qquad\qquad \exists z.h[p] = \texttt{Some}\ z\ \land\ \mathcal{Q}\ s\{z/rt\} \\
\mid \ldots \\
\textbf{end}
\end{array}$$

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

$$\frac{}{\{\mathcal{WP}\ i\ Q\}\, i\, \{\mathcal{Q}\}}$$

$$\frac{\{\mathcal{P}\}\, c_1\, \{\mathcal{R}\} \quad \{\mathcal{R}\}\, c_2\, \{\mathcal{Q}\}}{\{\mathcal{P}\}\, c_1\, ; c_2\, \{\mathcal{Q}\}} \qquad \frac{\{\lambda s.\mathcal{P} \land \llbracket b \rrbracket_s\}\, c\, \{\mathcal{P}\}}{\{\mathcal{P}\}\, \texttt{while}\ b\ c\, \{\lambda s.\mathcal{P} \land \neg\llbracket b \rrbracket_s\}}$$

$$\frac{\{\lambda s.\mathcal{P} \land \llbracket b \rrbracket_s\}\, c_1\, \{\mathcal{Q}\} \quad \{\lambda s.\mathcal{P} \land \llbracket b \rrbracket_s\}\, c_2\, \{\mathcal{Q}\}}{\{\mathcal{P}\}\, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{\mathcal{Q}\}} \qquad \frac{\mathcal{P} \to \mathcal{P}' \quad \{\mathcal{P}'\}\, c\, \{\mathcal{Q}'\} \quad \mathcal{Q}' \to \mathcal{Q}}{\{\mathcal{P}\}\, c\, \{\mathcal{Q}\}}$$

B Formalization of the Compositional Hoare Logic of [SU07]

As suggested in [SU07], we introduce predicate transformers that enforce assertions to be satisfiable for labels inside (resp. outside) a particular domain:

$$\mathcal{P}|_d \overset{def}{=} \lambda l.\mathcal{P}\ l \land l \in d \qquad\qquad \mathcal{P}|_{\overline{d}} \overset{def}{=} \lambda l.\mathcal{P}\ l \land l \notin d$$

Using above predicate transformers and the weakest-precondition function of the previous section (Sect. A.3), we formalize the rules for the compositional Hoare logic of [SU07] as follows:

$$\frac{}{\left[\begin{array}{ll}\lambda pc.\lambda s. & pc = l \land \mathcal{WP}\ c\ (\mathcal{P}\ (l+1))\ s\ \lor \\ & pc \neq l \land \mathcal{P}\ pc\ s\end{array}\right] l : c\, [\mathcal{P}]}$$

$$\frac{}{\left[\begin{array}{ll}\lambda pc.\lambda s. & pc = l \land (\mathcal{P}\ j\ s \lor j = l)\ \lor \\ & pc \neq l \land \mathcal{P}\ pc\ s\end{array}\right] l : \texttt{jmp}\ j\, [\mathcal{P}]}$$

$$\frac{}{\left[\begin{array}{ll}\lambda pc.\lambda s. & pc = l \land (\neg\llbracket b \rrbracket_s \land \mathcal{P}\ (l+1)\ s \lor \llbracket b \rrbracket_s \land (\mathcal{P} j\ s \lor j = l))\ \lor \\ & pc \neq l \land \mathcal{P}\ pc\ s\end{array}\right] l : \texttt{cjmp}\ b\ j\, [\mathcal{P}]}$$

$$\frac{\left[\mathcal{P}|_{\mathrm{dom}(c_1)}\right] c_1\, [\mathcal{P}] \quad \left[\mathcal{P}|_{\mathrm{dom}(c_2)}\right] c_2\, [\mathcal{P}]}{[\mathcal{P}]\, c_1 \oplus c_2 \left[\mathcal{P}|_{\overline{\mathrm{dom}(c_1 \oplus c_2)}}\right]}$$

$$\frac{}{[\mathcal{P}]\, nop\, [\mathcal{P}]} \qquad \frac{\forall l.\mathcal{P}\ l \to \mathcal{P}'\ l \quad [\mathcal{P}']\, c\, [\mathcal{Q}'] \quad \forall l.\mathcal{Q}'\ l \to \mathcal{Q}\ l}{[\mathcal{P}]\, c\, [\mathcal{Q}]}$$

C Compilation for Assembly: From While-loops to Jumps

Here follows the formalization of the compilation predicate of Sect. 4.4. The purpose of compilation here is to translate structured branching and while-loops into labelled conditional and unconditional jumps.

$$\frac{}{i \;{}^{l}\!\searrow_{l+1}\; l : i} \qquad \frac{c_1 \;{}^{l}\!\searrow_{l_1}\; c_1' \quad c_2 \;{}^{l_1}\!\searrow_{l_2}\; c_2'}{c_1 \,;c_2 \;{}^{l}\!\searrow_{l_2}\; c_1' \oplus c_2'}$$

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

$$\frac{c \;{}^{l+1}\!\searrow_{l_1}\; c'}{\texttt{while}\ b\ c \;{}^{l}\!\searrow_{l_1+1}\; l : \texttt{cjmp}\ (\neg b)\ (l_1+1) \oplus (c' \oplus l_1 : \texttt{jmp}\ l)}$$

D Technical Aspects of the Coq Formalization

D.1 Formalization of Programs, Semantics, and Logics

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] and of our mechanized formalization of [SU07] in the Coq proof-assistant.

We needed to substantially revise and extend our previous work [AM06] to address scalability issues. In our previous work, we were able to successfully verify assembly programs of up to 36 instructions but it was unclear whether the approach would scale. Since our implementation of BBS would anyway be much larger, we spent beforehand some time to rework most lemmas and tactics. We do not comment extensively about this part of the formalization 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 example, proof scripts of experiments carried out in [AM06] shrank by 70% in terms of lines of code).

The new aspect of our framework regarding formalization of assembly programs is the formalization of the proof-carrying code framework of [SU07]. As already explained in Sect. 4, we instantiate this framework to Separation Logic and MIPS instructions and extend it to deal with error-states. The table below 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):

Reference in [SU07]	Reference in [Code] and status	Proof script size
Section 2	file `goto.v`	460 lines
Figure 1, Lemma 1, 3	Done	
Lemma 2	Particular cases only	
Section 3	file `sgoto.v`	745 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`	1429 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`	371 lines
Section 5	Not done	
Appendix A	Done (revision of [AM06])	
	file `state.v`	615 lines
	file `mips.v`	881 lines
	file `mips_hoare.v`	1045 lines
Appendix B		
Theorems 6–7, 15–18	Done (spread over above files)	

D.2 Formal Proof of the BBS Triple

The formal proof of the Separation Logic triple of Sect. 5.1 is technically the most demanding part of the proof effort. Indeed, we are dealing with a large assembly program (at least by the current standards of proof assistant-based verification, according to related work dedicated to formal verification of functional correctness of assembly programs, e.g., [AM06, MG07]). Our assembly implementation of BBS is 239 instructions long and spread over several snippets of code. The table below makes it more precise which snippets are used and their respective size:

Function	Reference in [Code]	Program size
BBS	`bbs_prg.v`	14 insns
Montgomery strict	`mont_mul_strict_prg.v`	9 insns
Montgomery raw	`mont_mul_prg.v`	36 insns
Multi-precision subtraction	`multi_sub_prg.v`	20 insns
Multi-precision comparison	`multi_lt_prg.v`	11 insns
Array initialization	`multi_zero_prg.v`	6 insns

(In the table above, "Montgomery raw" stands for the Montgomery multiplication verified in [AM06], and "Montgomery strict" is the Montgomery multiplication extended with initialization, comparison, and subtraction, so as to implement modular multiplication, see Appendix E.)

The table below summarizes the size of proof scripts used in the proof of the Separation Logic triple of BBS (Sect. 5.1). 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 [AM06, MG07], we think that 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—file `mont_exp_triple.v` in [Code]—, not used in this paper though). We cannot compare about BBS itself since it is a new contribution of the work in this paper:

Function	Reference in [Code]	Size
BBS	`bbs_triple.v`	841 lines
Montgomery strict	`mont_{mul,square}_strict_init_triple.v`	601 lines
Montgomery raw	`mont_{mul,square}_triple.v`	1205 lines
Multi-precision subtraction	`multi_sub_inplace_left_triple.v`	506 lines
Multi-precision comparison	`multi_lt_triple.v`	405 lines
Array initialization	`multi_zero_triple.v`	129 lines
Total		3687 lines

(In the table above, formal proofs of the Montgomery multiplication are duplicated: one version for multiplication and one version for squaring. This is because, even though the same program can be used to perform multiplication and squaring, Separation Logic triples are slightly different: this stems from our use of the separating conjunction. As a result, there are two formal proofs but so similar in contents that we just display the size of only one of them.)

E Implementation of BBS in Assembly: Modular Multiplication

In this section, we explain the concrete implementation of multi-precision square modulo used in the assembly code of BBS (Sect. 2.2).

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 highly-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 not almost a multiplication modulo except for the parasite value β^k and because $Z \not< M$ in general. To turn the Montgomery multiplication 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 $Z' = X.Y \pmod{M}$.

Here follows the assembly code for the Montgomery multiplication extended with comparison and subtraction. It makes use of the functions montgomery (the function certified in [AM06]), (in-place) subtraction multi_sub (derived from [AM06]), multi-precision comparison multi_lt and an initialization function multi_zero (see [Code] for the details):

```
mont_mul_strict_init $\stackrel{def}{=}$
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 81                 (* 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
80:      jmp 118
81:      multi_lt_prg k z m X Y int ext Z M
93:      beq int gpr_zero 96      (* is the output bigger than the modulus? *)
94:        skip
95:        jmp 118
96:        multisub k one z m z ext int quot C Z X Y X
118:
```

Using the assembly code just above, we can complete the assembly code of BBS given in Sect. 2.2. Let us assume that the register $b2k$ points to the pre-computed k-word $A = \beta^{2k} \pmod{M}$. Then the following two successive calls to mont_mul_strict_init performs a square modulo:

```
mul_mod $\overset{def}{=}$
6:    mont_mul_strict_init k alpha x x y m one ext int X B2K Y M quot C t s
118:  mont_mul_strict_init k alpha y b2k x m one ext int X B2K Y M quot C t s
320:
```