Certifying Assembly with Formal Security Proofs: the Case of BBS[☆]

Reynald Affeldt[a,*], David Nowak[a], Kiyoshi Yamada[a,b]

[a] *National Institute of Advanced Industrial Science and Technology (AIST), Central 2, 1-1-1 Umezono, Tsukuba, Ibaraki, 305-8568 Japan*
[b] *Present address: Lepidum Co., Ltd., Village Sasazuka III Bldg 6F, 1-30-3 Sasazuka, Shibuya-ku, Tokyo, 151-0073 Japan*

Abstract

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

Keywords: Hoare logic, Assembly language, Coq, PRNG, Provable security

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 ad-

[☆]A preliminary version of this work appeared in the proceedings of the 9th International Workshop on Automated Verification of Critical Systems [ANY09].

[*]Corresponding author

Email address: `reynald.affeldt@aist.go.jp` (Reynald Affeldt) vanced algorithms with low-level implementations in assembly language. This already makes formal proof technically difficult.

(b) Security guarantees about cryptographic primitives is the matter of *security proofs*, as practiced by cryptographers. In essence, these proofs aim at showing the security of cryptographic primitives by reduction to computational assumptions. Formal proofs of such reductions also involve probability theory or group theory.

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

In this article, we address the issue of formal verification of cryptographic assembly code with security proofs. As pointed out above, formal verification of cryptographic assembly code and formal verification of security proofs are not the same matter, even though both deal with cryptography. As an evidence of this mismatch, one can think of a cryptographic function such as encryption: its security proof typically relies on a high-level mathematical description, but when laid down in terms of assembly code such a function exhibits restrictions due to the choice of implementation. We are therefore essentially concerned about the integration of these two kinds of formal proofs. We do not question here the theoretical feasibility of such an integration; rather, we investigate its practical aspects when formal verification is carried out within a proof assistant based on proof theory.

Indeed, proof assistants based on proof theory, such as Coq [Coq10] or HOL, can be regarded as privileged tools when it comes to formal verification of security properties. They implement higher order logics that are expressive enough to model and reason about advanced mathematics as well as precise computation models, and their trusted computing base being small and well-understood provides adequate reliability. With such tools, formal verification is by default performed interactively (by means of *tactics*) but interaction can be automated and intermediate results can be aggregated (in the form of libraries of *lemmas*) to facilitate other formal verifications. As a consequence, various frameworks for formal verification of cryptography using proof assistants based on proof theory have already been proposed: on the one hand, [AM06, MG07] for cryptographic assembly code, and on the other hand, [ATM07, BBU08, BGZ09, Now07] for security proofs. However, it is not clear how to connect them in practice, or, in other words, how the formal security proof for a cryptographic primitive relates to its formally-verified implementation. Whatever connection is to be provided between two such frameworks, it has to be developed in a clear way, both understandable by cryptographers and implementers, and in a reusable fashion, so that new verification efforts can build upon previous ones.

Our main contribution is to propose a concrete approach, supported by a reusable formal framework on top of the Coq proof assistant [Coq10], for verification of assembly code together with security proofs. As a concrete evidence of usability, we formally verify a pseudorandom number generator written in assembly for smartcards with a proof of unpredictability. This choice of application is not gratuitous: this is the first step before verifying more cryptographic primitives, since many of them actually rely on pseudorandom number generation. To achieve our goal, we extend and integrate two existing frameworks developed for the Coq proof assistant: one for formal verification of assembly code [AM06], and another for formal verification of cryptographic primitives [Now07]. More precisely, our technical contributions consist in the following:

- We propose an integration in terms of *game-playing* [Sho04], a popular setting to represent security proofs. We introduce a new kind of game transformation to serve as a bridge between assembly code and algorithms as dealt with by cryptographers. This allows for a clear integration, that paves the way for a modular framework, understandable by both cryptographers and implementers.

- We extend the formal framework for assembly code of [AM06] to connect with the formal framework for security 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 logic proofs.

- We provide the first assembly program for a pseudorandom number generator that is formally verified with a security proof. The generator in question is the Blum-Blum-Shub pseudorandom number generator (hereafter, BBS) [BBS86] that we implement in the SmartMIPS assembly.

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 security 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 security proof of BBS in assembly, from its proof of functional correctness to the implementation step that enables integration with the security proof from [Now08]. In Sect. 6, we validate our formalization by producing automatically a SmartMIPS binary and experimenting its execution. In Sect. 7, we comment on technical aspects of the Coq formalization. In Sect. 8, we comment on related work, and conclude and comment on future work in Sect. 9.

2. The BBS Pseudorandom Number Generator

2.1. The BBS Algorithm

The security of the Blum-Blum-Shub pseudorandom number generator [BBS86] is based on mathematical properties of integers modulo. We recall some basic notions of number theory: an integer is a quadratic residue modulo m if it is congruent to a square modulo m; the Legendre of an integer (modulo a prime number) is $+1$ if it is a quadratic residue, and -1 otherwise; the Jacobi of an integer modulo m is the product of its Legendres for each prime factor of m. BBS exploits the *quadratic residuosity problem*: the problem of deciding for $\mathbb{Z}_m^*$ (the multiplicative group of integers modulo m), where m is the product of two distinct odd primes, whether elements with Jacobi $+1$ are in

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

The *quadratic residuosity assumption* is the assumption that this problem is intractable ([DK07], ch. 6). BBS exploits the quadratic residuosity assumption in the particular case of m being a Blum integer, i.e., the product of two distinct odd primes congruent to 3 modulo 4. Informally, the reason why BBS is secure is that a successful attack on BBS would make the quadratic residuosity problem tractable, and thus contradict the assumption that it is not.

Here follows an implementation of BBS as a Coq function. It performs iteratively squaring modulo and outputs the result of parity tests:

```
bbs $(len : \mathbb{N})$ $(seed : \mathbb{Z}_m^*)$ $\overset{def}{=}$ bbs_rec $len$ $seed^2$
bbs_rec $(len : \mathbb{N})$ $(x : QR_m)$ $\overset{def}{=}$
  match $len$ with
  | $0 \Rightarrow []$
  | $len' + 1 \Rightarrow$ parity $x$ :: bbs_rec $len'$ $x^2$
  end
```

The input is the desired number of pseudorandom bits *len* and a random *seed* for initialization.

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. BBS can be proved *left-unpredictable* (hereafter, unpredictable) under the assumption that the quadratic residuosity problem is intractable (this is equivalent to proving that BBS passes all polynomial-time statistical tests [Yao82]).

2.2. Implementation of BBS in Assembly

The assembly code bbs_asm in Fig. 1 implements BBS. First, let us explain the notations in this figure. Instructions (in typewriter font) represent SmartMIPS instructions, a superset of MIPS32 with additional instructions for

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

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

smartcards [Mips01]. Each instruction is uniquely labeled (labels appear on the left and in branching instructions in typewriter font). Constants are indexed with their length in bits (e.g., 0_{16} stands for 0 represented as a half-word). The names of registers (in italic font) are parameters; only the null register `reg_zero` is hardwired in the program.

When the assembly code bbs_asm is called, the registers l, n, x, m, k, and r_{32} are expected to be initialized as follows. l contains the address of the storage space for the output (the pseudorandom number to be produced) and n contains the size in words of this storage space. x and m contain addresses to multi-precision integers (respectively, the integer to be squared and the modulus) and k contains the size in words minus one of these integers (in other words, the storage spaces are one word larger than the value contained in k, the reason for this is explained in Sect. 2.3). r_{32} contains the value 32, the word-size in bits.

The assembly code bbs_asm in Fig. 1 essentially consists of a loop with a nested loop. It starts by initializing the register i (a counter for the outer loop) and the register L (a pointer to the output) (labels 0 and 1).

The outer loop (from label 2 to label 237) produces pseudorandom words and stores them adequately. Each iteration starts by initializing the register j (a counter for the inner loop) and the register w (a temporary word of storage) (labels 3 and 4). After execution of the inner loop (explained below), the contents of w are stored in memory (label 234) and the pointer to the output as well as the outer loop counter are incremented (labels 235 and 236).

The inner loop (from label 5 to label 233) produces one word of pseudo-random bits. First, it first performs a in-place square modulo (mul_mod is an inlined assembly program explained in the next section) (label 6). Second, it stores the parity bit of the result into register w' (labels 228 and 229). Third, the parity bit is shifted and stored into w in an appropriate position using bitwise operations (labels 230 and 231). Last, the inner loop counter is incremented (label 232).

We refer the reader to the Coq development ([ANY11], file `mips_cmd.v`) for the formal definition of the semantics of each instruction.

2.3. Implementation of Modular Multiplication in Assembly

In this work, we implement multi-precision square modulo using two successive Montgomery multiplications [Mon85], as explained below in more details. This is an alternative to classical modular multiplication that is implemented using a multi-precision multiplication followed by a multi-precision division ([MOV01], ch. 14). Using the Montgomery multiplication is a reasonable way to implement multi-precision multiplication modulo: the complexity is quadratic like classical modular multiplication ([MOV01], ch. 14), it avoids the costly multi-precision division while benefiting from specifically-tailored instructions in cryptography-enhanced architectures such as SmartMIPS [Mips01]. 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 factor β^k in front of Z 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 $Z \geq M$ and (2) a second pass to eliminate the factor β^k in front of Z. This second pass requires as an additional input a k-word $B = \beta^{2k} \pmod{M}$; given Z such that $\beta^k Z = X.Y \pmod{M}$ (as provided by the first Montgomery multiplication), it suffices to compute Z' such that $\beta^k Z' = Z.B \pmod{M}$: if M is odd (this is generally the case for cryptographic applications), one obtains as desired $Z' = X.Y \pmod{M}$. Algorithms in pseudo-code and illustrations can be found in [MOV01], ch. 14; [AM06] provides a concrete implementation of the Montgomery multiplication together with its formal verification.

The assembly code mont_mul_strict_init in Fig. 2 implements the Montgomery multiplication extended with comparison and subtraction, as explained in the previous paragraph. When the assembly code mont_mul_strict_init is called, registers x, y, m, z, k, and $alpha$ are expected to be initialized as follows. x, y, m contain addresses to multi-precision integers (respectively, the integers to be multiplied and the modulus). z contains the address of the storage space for the output. k contains the size in words minus one of the storage spaces pointed to by the previous registers; the storage spaces are one word larger than the value

```
mont_mul_strict_init $\overset{def}{=}$
  6:  multi_zero $r_e$ $k$ $Z$ $z$                         (* output initialization *)
 13:  mflhxu reg_zero                               (* multiplier initialization *)
 14:  mthi reg_zero
 15:  mtlo reg_zero
 16:  montgomery $k$ $alpha$ $x$ $y$ $z$ $m$ $r_1$ $r_e$ $r_i$ $X$ $Y$ $M$ $Z$ $q$ $C$ $t$ $s$
 54:  beq $C$ reg_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 $r_e$ $r_1$ $z$ $m$ $z$ $M$ $r_i$ $q$ $C$ $Z$ $X$ $Y$ $X$
 78:    jmp 117
 79:    multi_lt_prg $k$ $z$ $m$ $X$ $Y$ $r_i$ $r_e$ $Z$ $M$
 94:    beq $r_i$ reg_zero 97   (* is the output bigger than the modulus? *)
 95:      nop
 96:      jmp 117
 97:      multisub $k$ $r_1$ $z$ $m$ $z$ $r_e$ $r_i$ $q$ $C$ $Z$ $X$ $Y$ $X$
117:
```

Figure 2: The Montgomery multiplication extended with comparison and subtraction

```
mul_mod $\overset{def}{=}$
  6:  mont_mul_strict_init $k$ $alpha$ $x$ $x$ $y$ $m$ $r_1$ $r_e$ $r_i$ $X$ $B$ $Y$ $M$ $q$ $C$ $t$ $s$
117:  mont_mul_strict_init $k$ $alpha$ $y$ $b$ $x$ $m$ $r_1$ $r_e$ $r_i$ $X$ $B$ $Y$ $M$ $q$ $C$ $t$ $s$
228:
```

Figure 3: Modular multiplication implemented by two Montgomery multiplications

contained in k to cope with the possibility for the Montgomery multiplication to produce outputs larger than the modulus. *alpha* contains the value α such that $m_0.\alpha = -1 \pmod{\beta}$ where m_0 is the least significant word of the modulus; this is a prerequisite for the correct execution of the Montgomery multiplication (see [MOV01], ch. 14).

The assembly code mont_mul_strict_init starts by zeroing the output storage (label 6) and the SmartMIPS multiplier, a set of special registers (labels 13, 14 and 15). These initializations to zero are a prerequisite for the correct execution of the rest of the code. Then, the program calls the Montgomery multiplication montgomery, the function verified in [AM06] (label 16).

As explained above, the result of the Montgomery multiplication may be larger than the modulus, in which case we need to perform an additional subtraction. Our implementation of the Montgomery multiplication is such that the value of the most significant word of the output appears in the register C; the execution of the rest of the program depends on its value (hence the test at label 54).

If the register C is not zero, then the result of the Montgomery multiplication is necessarily larger than the modulus, and thus we have to perform a subtraction. In this case, we store the value of C appropriately into memory (labels 55 and 56) and performs a in-place multi-precision subtraction (label 58).

If the register C is zero, then we need to perform a (multi-precision) comparison to determine whether the result of the Montgomery multiplication is larger than the modulus (label 79). According to the result of this comparison (label 94), a subtraction may be necessary. If the result is smaller than the modulus, then we are done (labels 95 and 96); otherwise, we perform a in-place multi-precision subtraction (label 97).

We refer the reader to the Coq development [ANY11] for the formalization of the functions montgomery (taken from [AM06]), in-place subtraction multi_sub (derived from [AM06]), multi-precision comparison multi_lt and the initialization function multi_zero.

The assembly code mul_mod in Fig. 3 implements modular multiplication by means of two successive calls to the extended version of the Montgomery multiplication (labels 6 and 117). It expects in particular that the register b points to the pre-computed k-word $B = \beta^{2k} \pmod{M}$ as explained previously.

3. Game-based Proofs for Assembly

Security proofs usually apply to algorithms without any consideration for implementation. In order to prove unpredictability directly on assembly code, we propose in Section 3.3 to lift a standard definition borrowed from *game-playing* [Sho04]. Game-playing is a methodology to write security 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 probabilistic program) to be solved by an attacker, the latter being modeled as a probabilistic polynomial-time (PPT) function whose code is unknown. A security 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 the corresponding game to some computational assumption through *game transformations.*

In this section, we illustrate the game-playing methodology and our extension with implementation steps in the case of unpredictability for BBS.

3.1. Unpredictability

Regarding unpredictability for a function f, the game unpredictability f is defined as follows (following the formalization of [Now08]): a seed is picked at random in the set of seeds ($\mathbb{Z}_m^*$ 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; and the latter returns its guess $\widehat{b_0}$. The result of the game is whether the guess is right or not. Using notations from [Now08], this game is written as:

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

The goal is then to prove that the above game is indistinguishable from the following game which simply consists in flipping a coin:

$$\mathsf{flip} \quad \overset{def}{=} \quad \left\{ \begin{array}{l} b \overset{R}{\leftarrow} \{true, false\}; \\ \textbf{return } b \end{array} \right.$$

3.2. Game transformations

The proof of indistinguishability consists in a sequence of game transformations from a game to another game that is indistinguishable. We illustrate this by showing the first game transformation in the proof of unpredictability for BBS (as presented in [Now08]). Following Shoup's classification, this transformation is a *bridging step*: it consists in restating how certain quantities can be computed in a completely equivalent way. We first unfold the definition of bbs in unpredictability bbs and obtain the game

$$\begin{array}{l} seed \overset{R}{\leftarrow} \mathbb{Z}_m^*; \\ [b_0, \ldots, b_{len}] \leftarrow \mathsf{bbs\_rec}(len+1, seed^2); \\ \widehat{b_0} \Leftarrow A([b_1, \ldots, b_{len}]); \\ \textbf{return } (\widehat{b_0} = b_0) \end{array}$$

Because the function which maps an $x \in \mathbb{Z}_m^*$ to $x^2 \in QR_m$ is a surjective four-to-one function, choosing at random a $seed \in \mathbb{Z}_m^*$ and then use only its square $seed^2$ is the same as picking at random an $x \in QR_m$ and using x instead of $seed^2$. The game can thus be rewritten as the indistinguishable game

$$\begin{array}{l} x \overset{R}{\leftarrow} QR_m; \\ [b_0, \ldots, b_{len}] \leftarrow \mathsf{bbs\_rec}(len+1, x); \\ \widehat{b_0} \Leftarrow A([b_1, \ldots, b_{len}]); \\ \textbf{return } (\widehat{b_0} = b_0) \end{array}$$

A series of such transformations based on various mathematical facts goes on until we reach the game

$$\begin{array}{l} x \overset{R}{\leftarrow} \mathbb{Z}_m^*(+1); \\ \widehat{b} \Leftarrow \left\{ \begin{array}{l} \widehat{b'} \Leftarrow A(m, x^2); \\ \textbf{return } (\widehat{b'} \oplus \mathsf{parity}(x) \oplus 1); \end{array} \right. \\ \textbf{return } (\widehat{b} = \mathsf{qr}(x)) \end{array}$$

where $\mathsf{qr}(x)$ is equal to *true* if x is a quadratic residue or *false* otherwise, and $\mathbb{Z}_m^*(+1)$ is the subset of $\mathbb{Z}_m^*$ whose elements have Jacobi +1.

This is at this point that we use the quadratic residuosity assumption that states that games of the form

$$\begin{array}{l} x \overset{R}{\leftarrow} \mathbb{Z}_m^*(+1); \\ \widehat{b} \Leftarrow A'(m, x); \\ \textbf{return } (\widehat{b} = \mathsf{qr}(x)) \end{array}$$

are indistinguishable (for any attacker A') from the game flip, i.e., the attacker cannot do better than flipping a coin when trying to guess the quadratic residuosity of an element of $\mathbb{Z}_m^*(+1)$. See [Now08] for the complete formalization in Coq. Following Shoup's classification, this is a *transition based on indistinguishability.* We have thus reduced the unpredictability game into a game consisting in flipping a coin. This concludes the security proof.

Note that there is a third kind of game transformation in the classification by Shoup: *transition based on failure events* [Sho04]. However it is not used in the security proof of BBS.

3.3. Implementation step

To define unpredictability for assembly code, one needs to lift the previous game definition because it applies to mathematical functions without any consideration for their implementations. This makes a difference because, contrary to mathematical functions, assembly code does not work as intended for arbitrary input, because of restrictions inherited from implementation choices. Our basic idea is thus to extract from the assembly code c its semantics in terms of a mathematical function and to inject it into the definition of the unpredictability game:

$$\mathsf{unpredictability\_assembly}\ c \ \overset{def}{=}\ \mathsf{unpredictability}\ \ (\lambda x.\mathsf{decode}\,([\![c]\!](\mathsf{encode}\ x))))$$

where $[\![c]\!]$ is the function that maps an initial state to the final state resulting from the execution of c (for $[\![c]\!]$ to be well-defined as a function, the assembly code c has to be deterministic and terminating); encode and decode respectively encode input data into an initial state and decode a final state into output data.

One needs to make clear under which restrictions the assembly code behaves as intended, i.e., state a predicate restrictions such that:

$$\forall x.\mathsf{restrictions}\ x \ \Rightarrow\ \mathsf{decode}\,([\![c]\!](\mathsf{encode}\ x)) = f\,x \quad (\textsc{Implem. Step Lemma})$$

where f is the mathematical function that c is supposed to implement. This correctness property allows to rewrite the game unpredictability_assembly c into the initial game unpredictability f of cryptographers' game-based security proof.

The advantage of the lifting explained above is that it makes clear how to organize formal verification of assembly code with security 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 kind of game transformations listed by Shoup [Sho04], this makes it easier to develop a formal framework for verification of assembly with security proofs: pick up a formal framework for game-based proofs and a formal framework for assembly, and add the machinery for implementation steps.

Fig. 4 gives an overview of a game-based security proof for an implementation. One starts with the game for implementation that includes interface functions encode and decode in order to exchange data between the computer world of the implementation (arrays of 32-bit words, etc.) and the mathematical world of the security definition (integers modulo, etc.). The first game transformation is the implementation step that replaces the implementation and its interface functions with the cryptographic algorithm. The correctness proof of the implementation ensures that the new game is indistinguishable from the previous one. Then the usual game-based security proof by cryptographers can be carried on until one reaches the intractable game that is thus proved indistinguishable from the initial one: if the attacker could win the implementation game, then it could also win the intractable game.

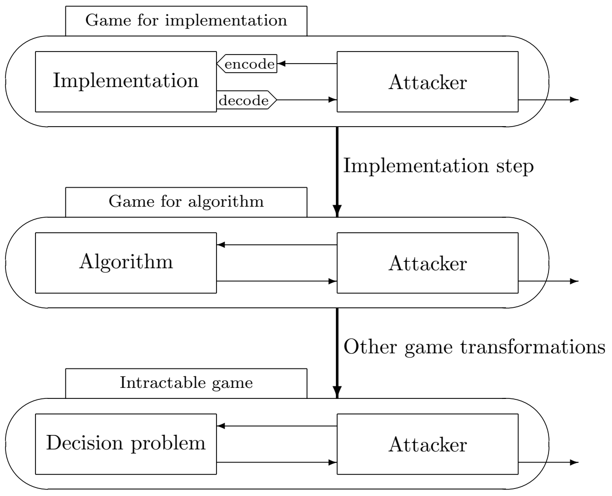

Figure 4: Overview of a game-based security proof with an implementation step

4. Verification of Functional Correctness of Assembly

To perform security 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 that are either complex [TA06] or verbose [SU07]. When used directly, these logics are thus less practical than standard Hoare logic for WHILE programs (i.e., imperative programs built out of while-loops). Yet, here, we choose to formalize the proof-carrying code framework of [SU07] because it 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 Hoare logic can be obtained by compilation from operational semantics and Hoare logic for WHILE programs. In other words, it becomes possible to work with WHILE programs while still being able to recover proofs in Hoare logic for assembly with jumps.

In this section, we formalize the proof-carrying code framework of [SU07] in a generic way as a Coq module parametrized by a formalization of Hoare logic for WHILE programs. This latter formalization of Hoare logic itself abstracts the implementation of states, the language of assertions, and the underlying set of one-step, non-branching instructions. As a consequence, we can instantiate the proof-carrying code framework of [SU07] with various concrete instruction sets and we are also able to accommodate the connectives of *separation logic*, as well as the frame rule [Rey02]. Indeed, this is separation logic that we use in Sect. 5.1 to specify and prove the functional correctness of the implementation of BBS in assembly.

This section is organized as follows. In Sect. 4.1, we explain the generic formalization of Hoare logic for WHILE. In Sections 4.2 and 4.3, we provide a generic formalization of the operational semantics and of the Hoare logic of SGOTO, the imperative language with jumps of [SU07]. In Sect. 4.4, we formalize the compilation relation between SGOTO and WHILE programs, as well as the relation between proofs in the Hoare logic of WHILE and proofs in the Hoare logic of SGOTO. In Sect. 4.5, we instantiate our generic formalization with a concrete instruction set, namely SmartMIPS [Mips01], and instrument it with the connectives of separation logic.

4.1. A Parametric Formalization of Hoare Logic

A state is a pair of a *store* and a *heap*: $\mathit{state} \overset{def}{=} \mathit{store} \times \mathit{heap}$. Since, we are dealing with low-level languages such as assembly whose instructions may trap, we distinguish error states by an option type: a state s can be the error state `None` or is otherwise noted $\lfloor s \rfloor$.

We assume a set of one-step, non-branching instructions i whose semantics is given by the predicate $s - i \rightarrow s'$ where i is an instruction, s (resp. s') is the state before (resp. after) its execution such that the following properties hold:

Parameter $\mathit{exec0\_deter} : \forall s, i, s_1, s_2 \,.\, s - i \rightarrow s_1 \;\Rightarrow\; s - i \rightarrow s_2 \;\Rightarrow\; s_1 = s_2$.
Parameter $\mathit{from\_none0} : \forall c, s \,.\, \texttt{None} - c \rightarrow s \;\Rightarrow\; s = \texttt{None}$.
Parameter $\mathit{cmd0\_terminate} : \forall c, s \,.\, \exists s' \,.\, \lfloor s \rfloor - c \rightarrow s'$.

At this point, it should be noted that we consider a set of deterministic instructions because we are dealing with assembly. The absence of deterministic instructions comes in contrast with standard separation logic [Rey02] that accommodates dynamic memory allocation ("malloc"). This will not prevent us from formalizing the connectives of separation logic (see Sect. 4.5) in such a way that the frame rule is provable (following [MAY06]).

Boolean expressions (ranged over by b) are equipped with an evaluation function $\mathcal{B}[\![b]\!]_s$ where s is a state. At this level of abstraction, we require the type of boolean expressions to feature at least a negation operator (to be used later for compilation) such that:

$$\textbf{Parameter}\ \textit{eval\_b\_neg} : \forall b, s\,.\ \neg\mathcal{B}[\![b]\!]_s \Leftrightarrow \mathcal{B}[\![\neg b]\!]_s.$$

The syntax of While programs (ranged over by c) is built out of the one-step, non-branching instructions i, sequences $(c_1\,\mathtt{;}\,c_2)$, structured branching ($\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$), and while-loops ($\mathtt{while}\ b\ c$). The standard big-step operational semantics is defined in Fig. 5.

$$\frac{s\ -\ i\ \rightarrow\ s'}{s\ -\ i\ \twoheadrightarrow\ s'} \qquad \frac{s - c_1 \twoheadrightarrow s'' \quad s'' - c_2 \twoheadrightarrow s'}{s - c_1\,\mathtt{;}\,c_2 \twoheadrightarrow s'} \qquad \frac{}{\mathtt{None}\ -\ c\ \twoheadrightarrow\ \mathtt{None}}$$

$$\frac{\mathcal{B}[\![b]\!]_s \qquad \lfloor s\rfloor - c_1 \twoheadrightarrow s'}{\lfloor s\rfloor - \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \twoheadrightarrow s'} \qquad \frac{\mathcal{B}[\![b]\!]_s \quad \lfloor s\rfloor - c \twoheadrightarrow s' \quad s' - \mathtt{while}\ b\ c \twoheadrightarrow s''}{\lfloor s\rfloor - \mathtt{while}\ b\ c \twoheadrightarrow s''}$$

$$\frac{\neg\mathcal{B}[\![b]\!]_s \qquad \lfloor s\rfloor - c_2 \twoheadrightarrow s'}{\lfloor s\rfloor - \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \twoheadrightarrow s'} \qquad \frac{\neg\mathcal{B}[\![b]\!]_s}{\lfloor s\rfloor - \mathtt{while}\ b\ c \twoheadrightarrow \lfloor s\rfloor}$$

Figure 5: Standard big-step operational semantics of While programs

The properties of states are specified using a *shallow embedding* of the logical connectives, i.e., assertions are functions from states (a pair of a store and a heap) to the type Prop of propositions in Coq: $\textit{assertion} \overset{def}{=} \textit{store} \Rightarrow \textit{heap} \Rightarrow \mathsf{Prop}$. Thanks to this embedding, it is possible to use the higher order logic of Coq to formalize various logical connectives. In fact, we use separation logic in Sect. 5.1 to specify and prove the functional correctness of the implementation of BBS in assembly. We defer to Sect. 4.5 explanations about how the connectives of separation logic are formalized using this embedding.

We assume defined for all instructions i, a relation $\{\!\{\mathcal{P}\}\!\}\,i\,\{\!\{\mathcal{Q}\}\!\}$, where $\mathcal{P}$ and $\mathcal{Q}$ are assertions, that is sound w.r.t. the operational semantics:

$$\begin{aligned}&\textbf{Parameter}\ \textit{soundness0} : \forall\mathcal{P},\mathcal{Q},i\,.\ \{\!\{\mathcal{P}\}\!\}\,i\,\{\!\{\mathcal{Q}\}\!\} \Rightarrow \forall s,h\,.\ \mathcal{P}\ s\ h \Rightarrow\\ &\quad\neg\,(\lfloor(s,h)\rfloor - i \twoheadrightarrow \mathtt{None}) \wedge (\forall s',h'\,.\ \lfloor(s,h)\rfloor - c \twoheadrightarrow \lfloor(s',h')\rfloor \Rightarrow \mathcal{Q}\ s'\ h')\,.\end{aligned}$$

In Fig. 6, we define a generic Hoare logic as the proof system for triples of the form $\{\mathcal{P}\}\,c\,\{\mathcal{Q}\}$ where $\mathcal{P}$ and $\mathcal{Q}$ are assertions and c is a While program.

Under the assumption *soundness0*, the proof system of Fig. 6 is proved sound w.r.t. the big-step operational semantics of Fig. 5.

As for relative completeness, let us define the weakest precondition as follows:

$$\begin{aligned}&\textit{wp\_semantics}\ c\ \mathcal{Q} \overset{def}{=} \lambda s,h\,.\ \neg\,(\lfloor(s,h)\rfloor - c \twoheadrightarrow \mathtt{None}) \wedge\\ &\quad(\forall s',h'\,.\ \lfloor(s,h)\rfloor - c \twoheadrightarrow \lfloor(s',h')\rfloor \Rightarrow \mathcal{Q}\ s'\ h')\,.\end{aligned}$$

$$\frac{\{\!\{P\}\!\}\, i\, \{\!\{Q\}\!\}}{\{\mathcal{P}\}\, i\, \{\mathcal{Q}\}} \qquad \frac{\{\mathcal{P}\}\, c_1\, \{\mathcal{R}\} \;\; \{\mathcal{R}\}\, c_2\, \{\mathcal{Q}\}}{\{\mathcal{P}\}\, c_1\, ; c_2\, \{\mathcal{Q}\}} \qquad \frac{\mathcal{P} \Rightarrow \mathcal{P}' \;\; \{\mathcal{P}'\}\, c\, \{\mathcal{Q}'\} \;\; \mathcal{Q}' \Rightarrow \mathcal{Q}}{\{\mathcal{P}\}\, c\, \{\mathcal{Q}\}}$$

$$\frac{\begin{array}{c}\left\{\lambda s, h \,.\, \mathcal{P}\ s\ h \wedge \mathcal{B}[\![b]\!]_{(s,h)}\right\} c_1\, \{\mathcal{Q}\} \\ \left\{\lambda s, h \,.\, \mathcal{P}\ s\ h \wedge \neg\mathcal{B}[\![b]\!]_{(s,h)}\right\} c_2\, \{\mathcal{Q}\}\end{array}}{\{\mathcal{P}\}\, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{\mathcal{Q}\}} \qquad \frac{\left\{\lambda s, h \,.\, \mathcal{P}\ s\ h \wedge \mathcal{B}[\![b]\!]_{(s,h)}\right\} c\, \{\mathcal{P}\}}{\{\mathcal{P}\}\, \texttt{while}\ b\ c\, \left\{\lambda s, h \,.\, \mathcal{P}\ s\ h \wedge \neg\mathcal{B}[\![b]\!]_{(s,h)}\right\}}$$

Figure 6: Standard Hoare logic for While programs

Then the above proof system is also proved relatively complete under the following assumption:

$$\textbf{Parameter}\ \textit{wp\_semantics\_sound0} : \forall i, \mathcal{Q}\,.\ \{\textit{wp\_semantics}\ i\ \mathcal{Q}\}\, i\, \{\mathcal{Q}\}\,.$$

4.2. Operational Semantics of SGoto

We provide a generic formalization of SGoto, the imperative language with jumps of [SU07].

To accommodate jumps, states are extended with a *label* (that represents the value of the program counter of the instruction being currently executed): $\textit{lstate} \stackrel{def}{=} \texttt{option}\ (\textit{label} \times \textit{state})$. The fact that we handle error states (as an option type) is a slight generalization w.r.t. [SU07].

An assembly program is formalized as a set of labeled instructions. The latter are either labeled one-step, non-branching instructions or jump instructions (unconditional jumps $\texttt{jmp}\ l$ or conditional jumps $\texttt{cjmp}\ b\ l$). $\text{dom}(c)$ is the set of the labels of the instructions of the assembly program c. Sets of labeled instructions are put together using $\oplus$.

The operational semantics of assembly programs is a predicate noted $s \succ c \rightharpoonup s'$ where c is a set of labeled instructions, s (resp. s') is the state before (resp. after) its execution. It is defined inductively by the rules of Fig. 7. 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.

4.3. Hoare Logic for SGoto

The non-standard operational semantics of the previous section gives rise to a non-standard but compositional Hoare logic for assembly with jumps [SU07].

The definition of assertions is extended so that satisfiability depends on the value of the current label: $\textit{assn} \stackrel{def}{=} \textit{label} \Rightarrow \textit{assertion}$.

A triple is noted $[\mathcal{P}]\, c\, [\mathcal{Q}]$ where $\mathcal{P}$ and $\mathcal{Q}$ are labeled 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: $\mathcal{P}|_d \stackrel{def}{=} \lambda l\,.\, \mathcal{P}\ l \wedge l \in d$, and $\mathcal{P}|_{\overline{d}} \stackrel{def}{=} \lambda l\,.\, \mathcal{P}\ l \wedge l \notin d$.

$$\frac{\lfloor s \rfloor \;-\; i \;\rightarrow\; \lfloor s' \rfloor}{\lfloor (l,s) \rfloor \;\succ\; l : i \;\rightarrowtriangle\; \lfloor (l+1, s') \rfloor} \qquad \frac{\lfloor s \rfloor \;-\; i \;\rightarrow\; \texttt{None}}{\lfloor (l,s) \rfloor \;\succ\; l : i \;\rightarrowtriangle\; \texttt{None}}$$

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

$$\frac{\mathcal{B}[\![b]\!]_s \quad l \neq l'}{\lfloor (l,s) \rfloor \succ l : \texttt{cjmp}\; b\; l' \rightarrowtriangle \lfloor (l', s) \rfloor} \qquad \frac{\begin{array}{c} l \in \mathrm{dom}(c_1) \\ \lfloor (l,s) \rfloor \succ c_1 \rightarrowtriangle s' \quad s' \succ c_1 \oplus c_2 \rightarrowtriangle s'' \end{array}}{\lfloor (l,s) \rfloor \;\succ\; c_1 \oplus c_2 \;\rightarrowtriangle\; s''}$$

$$\frac{\neg\mathcal{B}[\![b]\!]_s}{\lfloor (l,s) \rfloor \succ l : \texttt{cjmp}\; b\; l' \rightarrowtriangle \lfloor (l+1, s) \rfloor} \qquad \frac{\begin{array}{c} l \in \mathrm{dom}(c_2) \\ \lfloor (l,s) \rfloor \succ c_2 \rightarrowtriangle s' \quad s' \succ c_1 \oplus c_2 \rightarrowtriangle s'' \end{array}}{\lfloor (l,s) \rfloor \;\succ\; c_1 \oplus c_2 \;\rightarrowtriangle\; s''}$$

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

Figure 7: Big-step operational semantics of assembly with jumps

[SU07] defines a Hoare logic for an archetypal low-level language with only assignment. To allow for more one-step, non-branching instructions, we introduce an abstract weakest-precondition function $\mathcal{WP}_{insn}\; i\; \mathcal{P}$ such that:

Parameter $\mathit{wp0\_no\_err} : \forall s, h, i, \mathcal{P}\,.\; \mathcal{WP}_{insn}\; i\; \mathcal{P}\; s\; h \;\Rightarrow\; \neg\lfloor (s,h) \rfloor - i \rightarrow \texttt{None}$.
Parameter $\mathit{exec0\_wp0} : \forall s, h, i, s', h'\,.\; \lfloor (s,h) \rfloor - i \rightarrow \lfloor (s', h') \rfloor \;\Rightarrow$
$\forall \mathcal{P}. \mathcal{WP}_{insn}\; i\; \mathcal{P}\; s\; h \Leftrightarrow \mathcal{P}\; s'\; h'$.

Using the function $\mathcal{WP}_{insn}$, we formalize the rules for the compositional Hoare logic in Fig. 8.

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

$$\frac{}{\left[\begin{array}{ll} \lambda pc, s, h. & pc = l \wedge \left(\begin{array}{l} \neg\mathcal{B}[\![b]\!]_{(s,h)} \wedge \mathcal{P}\; (l+1)\; s\; h \vee \\ \mathcal{B}[\![b]\!]_{(s,h)} \wedge (\mathcal{P}\; j\; s\; h \vee j = l) \end{array}\right) \vee \\ & pc \neq l \wedge \mathcal{P}\; pc\; s\; h \end{array}\right] l : \texttt{cjmp}\; b\; j\, [\mathcal{P}]}$$

$$\frac{}{[\mathcal{P}]\, \texttt{nop}\, [\mathcal{P}]} \qquad \frac{}{\left[\begin{array}{ll} \lambda pc, s, h. & pc = l \wedge \mathcal{WP}_{insn}\; i\; (\mathcal{P}\; (l+1))\; s\; h \vee \\ & pc \neq l \wedge \mathcal{P}\; pc\; s\; h \end{array}\right] l : i\, [\mathcal{P}]}$$

$$\frac{\left[\mathcal{P}|_{\mathrm{dom}(c_1)}\right] c_1\, [\mathcal{P}] \quad \left[\mathcal{P}|_{\mathrm{dom}(c_2)}\right] c_2\, [\mathcal{P}]}{[\mathcal{P}]\, c_1 \oplus c_2 \left[\mathcal{P}|_{\overline{\mathrm{dom}(c_1 \oplus c_2)}}\right]} \qquad \frac{\begin{array}{c} \forall l. \mathcal{P}\; l \;\Rightarrow\; \mathcal{P}'\; l \\ [\mathcal{P}']\, c\, [\mathcal{Q}'] \\ \forall l. \mathcal{Q}'\; l \;\Rightarrow\; \mathcal{Q}\; l \end{array}}{[\mathcal{P}]\, c\, [\mathcal{Q}]}$$

Figure 8: Hoare logic for assembly with jumps

Using only the constructs and properties introduced so far in this section and following [SU07], the logic of Fig. 8 is formally proved sound and complete w.r.t. the non-standard big-step operational semantics of Fig. 7.

4.4. Compilation from While *to* SGoto

The derivations for the previous non-standard operational semantics and Hoare logic can also be obtained from standard operational semantics and Hoare logic for While programs through compilation [SU07]. 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, the non-standard Hoare logic of [SU07] being more verbose) 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 defined in Fig. 9 inductively by the predicate $c \;{}^{l}\!\searrow_{l'}\; c'$ where l (resp. l') is the start (resp. end) label of the compiled program.

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

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

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

Figure 9: Compilation from While to SGoto programs

Through compilation, derivations of operational semantics are transformed from the standard big-step operational semantics (Fig. 5) to the big-step operational semantics of assembly with jumps (Fig. 7). Similarly, proofs in separation logic are transformed from the standard ones (Fig. 6) to the ones of [SU07] (Fig. 8). This is captured by the following lemmas:

Lemma *preservation_of_evaluations* : $\forall c, s, l, c', s', l'$.
$c \;{}^{l}\!\searrow_{l'}\; c' \;\Rightarrow\; \lfloor s \rfloor \;-\; c \;\rightharpoonup\; \lfloor s' \rfloor \;\Rightarrow\; \lfloor (l, s) \rfloor \;\succ\; c' \;\rightharpoonup\; \lfloor (l + \mathrm{card}(\mathrm{dom}(c')), s') \rfloor$.

Lemma *preservation_hoare* : $\forall \mathcal{P}, \mathcal{Q}, c\,.\; \{\mathcal{P}\}\, c\, \{\mathcal{Q}\} \;\Rightarrow$
$\forall l, c', l'\,.\; c \;{}^{l}\!\searrow_{l'}\; c' \;\Rightarrow\; [\lambda pc, s, h\,.\, pc = l \wedge \mathcal{P}\; s\; h]\, c'\, [\lambda pc, s, h\,.\, pc = l' \wedge \mathcal{Q}\; s\; h]$.

4.5. Instantiation of While *and* SGoto *to SmartMIPS*

We instantiate the previous formalization of [SU07] to the SmartMIPS assembly language. Instantiation amounts to providing the definition of SmartMIPS states, the set of SmartMIPS instructions, following the official MIPS documentation [Mips01], as well as proofs for the assumptions about them introduced so far in this section. Instantiation provides modules for While and SGoto operational semantics, Hoare logic, and certified compilation from While to SGoto.

A SmartMIPS 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: $\mathcal{R}[\![r]\!]_s$ 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. $h_1 \perp h_2$ holds when h_1 and h_2 are disjoint and $h_1 \cup h_2$ represents the union of h_1 and h_2; $[p, z]$ represents a singleton heap that associates address p with the value z.

We give the semantics of one-step, non-branching instructions by defining as an inductive type a predicate noted $s\ -\ i\ \rightarrow\ s'$ where i is a SmartMIPS 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: $(v)_{int_{16} \rightarrow int_{32}}$ sign-extends the value v from 16 to 32 bits, $(v)_{int_{32} \rightarrow \mathbb{N}}$ interprets the value v as an unsigned integer, etc.

Figure 10 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.)

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

$$\frac{\forall p.\ \left(\mathcal{R}[\![base]\!]_s +_h (\mathit{off})_{int_{16} \rightarrow int_{32}}\right)_{int_{32} \rightarrow \mathbb{N}} \neq 4 \times p\ \vee\ h[p] = \texttt{None}}{\lfloor (s,h) \rfloor\ -\ \texttt{lw}\ rt\ \mathit{off}\ base\ \rightarrow\ \texttt{None}}\ \textsf{exec0\_lw\_error}$$

Figure 10: Semantics of `lw`

Accordingly, we also instantiate the weakest precondition function $\mathcal{WP}_{insn}$.

Here is an excerpt of this function for the "load word" instruction:

$$\begin{array}{l}
\mathcal{WP}_{insn}\ i\ \mathcal{Q} \overset{def}{=} \textbf{match}\ i\ \textbf{with} \\
\quad \ldots \\
\quad |\ \texttt{lw}\ rt\ \mathit{off}\ base \Rightarrow \lambda s, h. \exists p. \left(\mathcal{R}[\![base]\!]_s +_h (\mathcal{R}[\![\mathit{off}]\!]_s)_{int_{16} \to int_{32}}\right)_{int_{32} \to \mathbb{N}} = 4 \times p\ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \exists z. h[p] = \lfloor z \rfloor\ \wedge\ \mathcal{Q}\ s\{z/rt\}\ h \\
\quad \ldots \\
\quad \textbf{end}
\end{array}$$

In total, we have formalized the semantics of 29 SmartMIPS one-step, non-branching SmartMIPS instructions, as well as 4 conditional jumps (`beq`, `bne`, `bltz`, `bgez`).

Formalization of Separation Logic. In addition to the properties provided by the formalization of [SU07], the formal verification of BBS in assembly in the next section (Sect. 5.1) also relies on a formalization of separation logic, including the frame rule, the key lemma that allows us to compose code snippets. This formalization is essentially taken from [AM06] and [MAY06] and is orthogonal to the formalization of [SU07] that we carried out in the previous sections. The formalization of the connectives of separation logic takes advantage of the fact that the assertions are shallow-embedded. For example, given two assertions $\mathcal{P}$ and $\mathcal{Q}$, the separating conjunction is conveniently formalized as follows: $\mathcal{P} \star \mathcal{Q} \overset{def}{=} \lambda s, h. \exists h_1, h_2. h_1 \perp h_2 \wedge h = h_1 \cup h_2 \wedge \mathcal{P}\ s\ h_1 \wedge \mathcal{Q}\ s\ h_2$. The "mapsto" connective of separation logic that specifies a singleton heap is formalized as follows: $e \mapsto e' \overset{def}{=} \lambda s, h\,.\ \exists p\,.\ (\mathcal{E}[\![e]\!]_s)_{int_{32} \to \mathbb{N}} = 4 \times p \wedge h = [p, \mathcal{E}[\![e']\!]_s]$ where e and e' are expressions made of registers, constants, and a few arithmetic operators and $\mathcal{E}[\![e]\!]_s$ represents the value of the expression e in store s. All the details can be found in the Coq development [ANY11].

5. Security Proof for BBS in Assembly

5.1. Functional Correctness

In practice, we conduct formal proof using separation logic for WHILE programs and obtain afterwards the desired triple for assembly with jumps by applying the preservation lemma *preservation_hoare* (Sect. 4.4). The formal verification effort therefore concentrates on the triple of Fig. 11 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 relation defined in Fig. 9). Let us explain this triple by commenting in turn about the auxiliary variables, the precondition, and the postcondition.

Auxiliary Variables. The first lines of Fig. 11 introduce a set of pairwise distinct registers. n_n and n_k record the size in words of, resp., the pseudorandom number to be produced and the other multi-precision integers involved in the computation. X, M, L, B, and Y are lists of int_{32} machine integers that encode

Lemma *bbs_triple* :

$$\forall i, L, l, n, j, r_{32}, k, \alpha, x, y, m, r_1, r_e, r_i, X, Y, M, q, C, t, r_s, r_b, B, w', w.$$
$$\mathsf{nodup}\,(i, L, l, n, j, r_{32}, k, \alpha, x, y, m, r_1, r_e, r_i, X, Y, M, q, C, t, r_s, r_b, B, w', w, \mathtt{reg\_zero}) \Rightarrow$$
$$\forall n_n, n_k, \mathsf{X}, \mathsf{M}, \mathsf{L}, \mathsf{B}, \mathsf{Y}\,.\ |\mathsf{L}| = n_n\ \wedge\ |\mathsf{X}| = |\mathsf{B}| = |\mathsf{M}| = |\mathsf{Y}| = n_k\ \Rightarrow$$
$$[\mathsf{X}]_\beta\,, [\mathsf{B}]_\beta < [\mathsf{M}]_\beta\ \wedge\ [\mathsf{B}]_\beta = \beta^{2\,n_k} \pmod{[\mathsf{M}]_\beta}\ \wedge\ \mathsf{odd}([\mathsf{M}]_\beta)\ \Rightarrow$$
$$\forall v_\alpha.\ \ (\mathsf{M}.0)_{int_{32}\to\mathbb{N}} \cdot (v_\alpha)_{int_{32}\to\mathbb{N}} = -1 \pmod{\beta}\ \Rightarrow$$
$$\forall v_x, n_x, v_y, n_y, n_m, v_m, n_b, v_b, v_l, n_l.\ (v_x)_{int_{32}\to\mathbb{N}} = 4\,n_x\ \wedge\ (v_y)_{int_{32}\to\mathbb{N}} = 4\,n_y \wedge$$
$$(v_m)_{int_{32}\to\mathbb{N}} = 4\,n_m\ \wedge\ (v_b)_{int_{32}\to\mathbb{N}} = 4\,n_b\ \wedge\ (v_l)_{int_{32}\to\mathbb{N}} = 4\,n_l\ \Rightarrow$$
$$4\,n_y + 4\,(n_k+1) < \beta\ \wedge\ 4\,n_x + 4\,(n_k+1) < \beta\ \wedge\ 4\,n_l + 4\,n_n < \beta\ \Rightarrow$$

$$\left\{\begin{array}{l}\lambda s, h.\,(\mathcal{R}[\![k]\!]_s)_{int_{32}\to\mathbb{N}} = n_k\ \wedge\ (\mathcal{R}[\![n]\!]_s)_{int_{32}\to\mathbb{N}} = n_n\ \wedge\ \mathcal{R}[\![x]\!]_s = v_x\ \wedge\ \mathcal{R}[\![y]\!]_s = v_y \wedge \\ \mathcal{R}[\![m]\!]_s = v_m \wedge \mathcal{R}[\![r_b]\!]_s = v_b \wedge \mathcal{R}[\![\alpha]\!]_s = v_\alpha \wedge \mathcal{R}[\![l]\!]_s = v_l \wedge (\mathcal{R}[\![r_{32}]\!]_s)_{int_{32}\to\mathbb{N}} = 32\ \wedge \\ (x \mapsto \mathsf{X};;0_{32}) \star (m \mapsto \mathsf{M};;0_{32}) \star (l \mapsto \mathsf{L}) \star (y \mapsto \mathsf{Y};;0_{32}) \star (r_b \mapsto \mathsf{B})\ s\ h\end{array}\right\}$$

$$\mathsf{bbs\_asm}_{\mathsf{decompile}}\ i\ L\ l\ n\ j\ r_{32}\ k\ \alpha\ x\ y\ m\ r_1\ r_e\ r_i\ X\ Y\ M\ q\ C\ t\ r_s\ r_b\ B\ w'\ w$$

$$\left\{\begin{array}{l}\lambda s, h\,.\ \exists \mathsf{X}, \mathsf{L}, \mathsf{Y}\,.\ |\mathsf{L}| = n_n\ \wedge\ |\mathsf{X}| = |\mathsf{Y}| = n_k\ \wedge\ (\mathcal{R}[\![k]\!]_s)_{int_{32}\to\mathbb{N}} = n_k \wedge \\ (\mathcal{R}[\![n]\!]_s)_{int_{32}\to\mathbb{N}} = n_n \wedge \mathcal{R}[\![x]\!]_s = v_x \wedge \mathcal{R}[\![y]\!]_s = v_y \wedge \mathcal{R}[\![m]\!]_s = v_m \wedge \mathcal{R}[\![r_b]\!]_s = v_b\ \wedge \\ \mathcal{R}[\![\alpha]\!]_s = v_\alpha\ \wedge\ \mathcal{R}[\![l]\!]_s = v_l\ \wedge\ (\mathcal{R}[\![r_{32}]\!]_s)_{int_{32}\to\mathbb{N}} = 32\ \wedge \\ (x \mapsto \mathsf{X};;0_{32}) \star (m \mapsto \mathsf{M};;0_{32}) \star (l \mapsto \mathsf{L}) \star (y \mapsto \mathsf{Y};;0_{32}) \star (r_b \mapsto \mathsf{B})\ s\ h\ \wedge \\ \mathsf{flatten}(\mathsf{map}\ \mathit{bits}\ \mathsf{L}) = \mathsf{bbs\_fun\_rec}\ (32\,n_n)\ ({[\mathsf{X}]_\beta}^2 \pmod{[\mathsf{M}]_\beta})\ [\mathsf{M}]_\beta\end{array}\right\}$$

Figure 11: Functional correctness of BBS in assembly

multi-precision integers and $|\cdot|$ is the notation for their length. More precisely, X encodes the random seed, M encodes the modulus, L is for the pseudorandom number to be produced, B encodes $\beta^{2\,n_k} \pmod{M}$ as explained in Sect. 2.3, and Y is for temporary storage (for the intermediate result of the modular multiplication). The fact that the modulus is odd is used in the modular multiplication as explained in Sect. 2.3. The introduction of a value v_α equal to the inverse modulo β of the least significant word of the modulus is specific to the Montgomery multiplication and has been explained in Sect. 2.3. The following alignment restrictions (values v_i that are multiple of 4) are imposed by SmartMIPS instructions that access memory. Finally, the inequalities guarantee that data fits in the (finite) memory of the computer.

Note that, as long as $4(4n_k + n_n + 2) < \beta$, n_n and n_k can be very large, n_k effectively covering lengths for which the quadratic residuosity problem is indeed difficult in practice. This is one desirable side-effect of our approach to precisely pinpoint the range of n_k. It is here that the restrictions imposed by implementation choices mentioned in Sect. 3 appear, for the above triple cannot be proved for arbitrary values of n_n and n_k.

Precondition. The precondition associates registers' values with auxiliary variables and specifies the initial state by way of a formula using connectives of separation logic. The separating conjunction explained in Sect. 4.5 is used to enforce disjointness of memory regions. The $\Mapsto$ connective is a generalization of the "mapsto" connective explained in Sect. 4.5: it maps an address to a list of values contiguous in memory instead of a single value. $\mathsf{X} ;; 0_{32}$ means that a trailing 0_{32} is allocated at the end of multi-precision integers: this is to support potential overflows during the modular multiplication, as explained in Sect. 2.3.

Postcondition. In the postcondition, the lists X, L, and Y are existentially quantified because they correspond to memory regions modified by execution. In particular, the final contents of L are specified to be the intended list of pseudorandom bits. Note that we are using a generalized version of the BBS algorithm (bbs_fun below takes the modulus m in $\mathbb{Z}$, whereas bbs in Sect. 2.1 uses the types $\mathbb{Z}_m^*$ and QR_m):

```
bbs_fun $(len : \mathbb{N})$ $(seed : \mathbb{Z})$ $(m : \mathbb{Z})$ $\overset{def}{=}$ bbs_fun_rec $len$ $(seed^2 \pmod{m})$ $m$
bbs_fun_rec $(len : \mathbb{N})$ $(x : \mathbb{Z})$ $(m : \mathbb{Z})$ $\overset{def}{=}$
  match $len$ with
  | $0 \Rightarrow []$
  | $len' + 1 \Rightarrow$ parity $x$ :: bbs_fun_rec $len'$ $(x^2 \pmod{m})$ $m$
  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 security proof).

5.2. Extraction of the Semantics of BBS in Assembly

For the rest of this section (i.e., for Sections 5.2 and 5.3), we assume given a set of pairwise distinct registers i, L, l, n, j, r_{32}, k, α, x, y, m, r_1, r_e, r_i, X, Y, M, q, C, t, r_s, r_b, B, w', and w.

Before proceeding to the extraction of the semantics of BBS in assembly, we specialize the triple of Sect. 5.1 by introducing two functions encode and decode: encode n k $seed$ m builds a state from the requested number n of pseudorandom 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 decode s is the list of pseudorandom bits stored in the state s. These functions hide many technical details of the generic triple of Sect. 5.1 but impose a specific memory layout depicted in Fig. 12.

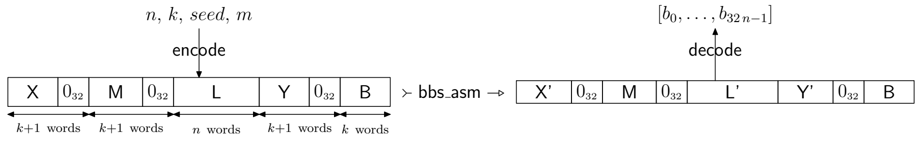

Figure 12: Encoding and decoding of input/output

Then the rule of consequence of Hoare logic allows for proving a simplified Hoare triple:

Lemma *bbs_triple_encode_decode* :
$\forall n_n, n_k, m, seed.\ 0 < m < \beta^{n_k} \wedge \mathsf{odd}(m) \wedge 0 \leq seed < m \wedge 4(4n_k + n_n + 2) < \beta \ \Rightarrow$
$\{\ \lambda s, h\,.\ \mathsf{encode}\ n_n\ n_k\ seed\ m = (s,h)\ \}$
$\mathsf{bbs\_asm}_{\mathsf{decompile}}\ i\ L\ l\ n\ j\ r_{32}\ k\ \alpha\ x\ y\ m\ r_1\ r_e\ r_i\ X\ Y\ M\ q\ C\ t\ r_s\ r_b\ B\ w'\ w$
$\{\ \lambda s, h\,.\ \mathsf{decode}\ (s,h) = \mathsf{bbs\_fun}\ (32\, n_n)\ seed\ m\ \}$

We now proceed to the extraction of the semantics of bbs_asm. As explained in Sect. 3, we prove for this purpose that bbs_asm is terminating. Determinism is a property of the assembly as it is formalized. First, we prove that there is a final state, without proving whether it is an error state or not:

Lemma *bbs_termination* : $\forall s_0.\exists s_f.$
$\lfloor s_0 \rfloor\ -\ \mathsf{bbs\_asm}_{\mathsf{decompile}}\ i\,L\,l\,n\,j\,r_{32}\,k\,\alpha\,x\,y\,m\,r_1\,r_e\,r_i\,X\,Y\,M\,q\,C\,t\,r_s\,r_b\,B\,w'\,w \ \twoheadrightarrow\ s_f.$

This is proved by induction on the variant of the outermost loop, and then on nested loops. Second, by the separation logic triple, we derive under the same restrictions the fact that this final state cannot be an error state. Then, using the *preservation_of_evaluations* lemma of Sect. 4.4, we obtain the following lemma:

Lemma *exec_bbs_asm* :
$\forall n_n, n_k, m, seed.\ 0 < m < \beta^{n_k} \wedge \mathsf{odd}(m) \wedge 0 \leq seed < m \wedge 4(4n_k + n_n + 2) < \beta \ \Rightarrow$
$\exists s_f.\lfloor (0, \mathsf{encode}\ n_n\ n_k\ seed\ m) \rfloor \succ$
$\mathsf{bbs\_asm}\ i\,L\,l\,n\,j\,r_{32}\,k\,\alpha\,x\,y\,m\,r_1\,r_e\,r_i\,X\,Y\,M\,q\,C\,t\,r_s\,r_b\,B\,w'\,w \ \twoheadrightarrow\ \lfloor (238, s_f) \rfloor.$

Because the above lemma is existential, it allows us to define a function $\mathsf{exec}_{\mathsf{bbs\_asm}}$ that realizes bbs_asm by mapping, in order, the number of desired 32 bits words for the pseudorandom bits, the number of 32 bits words necessary to encode the seed and the modulus, and finally the *seed* and the modulus m themselves so that the semantics of the assembly program bbs_asm is defined as follows (the function $\|\cdot\|$ returns the number of digits of integers, as formalized by the function `N_digits` of the standard Coq library):

$$[\![\mathsf{bbs\_asm}]\!]_{\mathsf{encode,decode}} \overset{def}{=} \lambda len, m, seed.\mathsf{prefix}_{len}\left(\mathsf{decode}\ \left(\mathsf{exec}_{\mathsf{bbs\_asm}}\ \left\lceil \frac{len}{32} \right\rceil\ \left\lceil \frac{\|m\|+1}{32} \right\rceil\ m\ seed\right)\right)$$

Since bbs_asm always returns a number of pseudorandom bits that is a multiple of 32, we need to take a prefix of the output because the unpredictability game from [Now08] may require less bits.

5.3. Implementation Step

The final step is to produce a rewriting lemma between the semantics of bbs_asm and the Coq function bbs (as explained in Sect. 3). First, using the separation logic triple of Sect. 5.1, by preservation of Hoare triples (lemma *preservation_hoare*) and the soundness of the Hoare logic of SGoto, we derive that:

Lemma *bbs_correct* :
$\forall n_n, n_k, m, seed.\ 0 < m < \beta^{n_k} \wedge \mathsf{odd}(m) \wedge 0 \le seed < m \wedge 4(4n_k + n_n + 2) < \beta \Rightarrow$
$\forall s_f, \lfloor(0, \mathsf{encode}\ n_n\ n_k\ seed\ m)\rfloor \succ$
$\quad \mathsf{bbs\_asm}\, i\, L\, l\, n\, j\, r_{32}\, k\, \alpha\, x\, y\, m\, r_1\, r_e\, r_i\, X\, Y\, M\, q\, C\, t\, r_s\, r_b\, B\, w'\, w \multimap \lfloor(238, s_f)\rfloor \Rightarrow$
$\mathsf{decode}\ s_f = \mathsf{bbs\_fun\_rec}\ 32\, n_n\ (seed^2 \pmod m))\ m.$

Finally, we are able to produce the rewriting lemma that lets us prove unpredictability of the assembly program using the formal proof of unpredictability of the Coq function as a sub-lemma:

Lemma *bbs_implem_step* : $\forall len, m, seed.\ 4\left(4\left\lceil\frac{len+1}{32}\right\rceil + \left\lceil\frac{\|m\|+1}{32}\right\rceil + 2\right) < \beta \Rightarrow$
$\quad [\![\mathsf{bbs\_asm}]\!]_{\mathsf{encode},\mathsf{decode}}\ (len + 1)\ m\ seed = \mathsf{bbs}\ (len + 1)\ seed.$

This last lemma can be used to perform the implementation step as described in Sect. 3, making use of the security proof of the BBS algorithm taken directly from [Now08].

6. Validating Experiment

As a validating experiment, we ran our implementation of BBS in a SmartMIPS simulator. We formalized in Coq a function that turns the abstract syntax for assembly into concrete syntax so as to produce snippets of code. The transformation relies on custom functions to turn registers, constant machine integers, etc. (`r2s`, `si2s`, etc. below) into Coq strings of the standard library of Coq. For illustration, here is the pattern matching corresponding to the load instruction `lw` whose formalization we explained in Sect. 4.5:

```
cmd0_to_string_aux c s  ≝  match c with
  ...
  | lw rt off base ⇒
    "lw " ++ (r2s rt (comma (si2s off (String "(" (r2s base (String ")" s))))))
  ...
  end.
```

To compare with the abstract syntax explained in Fig. 1, the transformation of bbs_asm into concrete syntax leads to a snippet of code that ends as follows:

```
...
L228: lw $t8, 0($t1)
L229: andi $t8, $t8, 1
L230: sllv $t8, $t8, $a1
L231: or $t9, $t9, $t8
L232: addiu $a1, $a1, 1
```

```
L233: b L5
L234: sw $t9, 0($v0)
L235: addiu $v0, $v0, 4
L236: addiu $at, $at, 1
L237: b L2
```

To run the above concrete syntax we append automatically constant initialization and loading instructions using a custom script.

The resulting assembly program is run thanks to a cross-compilation environment built out of GNU tools. Binaries can be executed in the GDB debugger that provides simulation for SmartMIPS. For illustration, let us run the resulting assembly program with input seed 31558851269462817406 and modulus 76318188169561490797 (= $8736028127 \times 8736028211$) (to be stored in 4-words multi-precision integers), and ask for pseudorandom 5×32 bits. Using appropriate `printf` statements, the final state of execution in GBD is rendered as follows:

```
X: 51494fd9 2550c923 00000000 00000000 00000000
M: 0421c56d 2320abca 00000004 00000000 00000000
L: bc84f41a eaa29f7a 632d1e04 38e58a1f 9eb00e50
Y: 8bc820b0 c8f2492f 00000003 00000000 00000000
B: c89a91c3 661fa03e 00000003 00000000
```

One can confirm that the memory area `L` displays the right result.

An interesting thing happened during this validating experiment that should be put to the credit of formal verification. In fact, the version of GDB (7.0.1) we used did not behave at first as expected, returning obviously a non-random output. After investigation, it turned out that the support for SmartMIPS simulation was not properly implemented. Thanks to the experience of having formalized the semantics of SmartMIPS in Coq, it took us little time to figure out the relevant bugs and provide an appropriate patch [Yam10].

7. 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 an extension and revision of previous work [AM06]. The revision was made necessary to address scalability issues. We do not comment extensively about it 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 (roughly, proof scripts of experiments in [AM06] shrank by 70% in terms of lines of codes).

The new aspect of our framework is the generic formalization of the proof-carrying code framework of [SU07], with its extension to support multiple instructions and error states, and its instantiation to SmartMIPS and separation logic. Table 1 makes it clear what is formalized[1] w.r.t. [SU07]; the size of proof scripts is given in terms of lines of codes, after removing blanks and comments.

[1]In brief, what we do not do: we do not formalize Section 5 of [SU07] and we formalize

Reference in [SU07]	Reference in [ANY11] and status	Size
Section 2	file `goto.v`	354 lines
Figure 1, Lemma 1, 3 Lemma 2	Done Particular cases only	
Section 3	file `sgoto.v`	579 lines
Section 3.1: Figure 2, Lemmas 4–5, Theorems 6–8, Corollary 9 *Section 3.2*: Figure 3, Theorem 10, Lemma 11, Theorem 12	Done Done	
Section 4	file `compile.v`	963 lines
Section 4.1: Figure 5, Lemmas 13–14, Theorems 15–16 *Section 4.2*: Theorems 17–18 *Section 4.3*	Done Done Done, file `sgoto_hoare.v`	293 lines
Section 5	Not done	
Appendix A	Done	
	file `while.v`	754 lines
Appendix B		
Theorems 6–7, 15–18	Done (spread over above files)	

Table 1: Formalization of [SU07]

The formal proof of the separation logic triple of Sect. 5.1 is 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]): 237 instructions (after inlining of all functions) that spread over several snippets of code. Table 2 gives the list of used snippets and their size.

Function	Reference in [ANY11]	Program size
BBS (Fig. 1)	`bbs_prg.v`	14 commands
Montgomery strict (Fig. 2)	`mont_mul_strict_prg.v`	9 commands
Montgomery raw [AM06]	`mont_mul_prg.v`	36 commands
Multi-precision subtraction [ANY11]	`multi_sub_prg.v`	18 commands
Multi-precision comparison [ANY11]	`multi_lt_prg.v`	13 commands
Array initialization [ANY11]	`multi_zero_prg.v`	6 commands

Table 2: The assembly code of BBS in Coq

Table 3 summarizes the size of proof scripts used in the proof of the separation logic triple of BBS. The Montgomery square is a variant of the Montgomery multiplication, so that proof scripts are similar in contents. The multi-precision subtraction works "in-place", i.e. it overrides the minuend with the difference.

only the so-called "non-constructive proofs" of Theorems 17 and 18 (indeed, for these two theorems, the proofs come in two flavors).

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 article and [AM06, MG07] (verification of multi-precision arithmetic and Montgomery multiplication—also Montgomery exponentiation, not used in this article but available online [ANY11]):

Function	Reference in [ANY11]	Size
BBS	`bbs_triple.v`	647 lines
Montgomery strict	`mont_mul_strict_init_triple.v`	502 lines
	`mont_square_strict_init_triple.v`	476 lines
Montgomery raw	`mont_mul_triple.v`	1025 lines
	`mont_square_triple.v`	997 lines
Multi-precision subtraction	`multi_sub_inplace_left_triple.v`	355 lines
Multi-precision comparison	`multi_lt_triple.v`	370 lines
Array initialization	`multi_zero_triple.v`	111 lines
Total		4483 lines

Table 3: Formal proof of the separation logic triple of BBS

Finally, Table 4 summarizes the size of proof scripts used to extract the semantics of BBS in assembly and to perform the implementation step as explained in Sect. 5.

Files contents	Reference in [ANY11]	Size
Termination proofs	`bbs_termination.v`	177 lines
	`mont_mul_strict_termination.v`	91 lines
	`mont_square_strict_termination.v`	92 lines
	`mont_mul_termination.v`	229 lines
	`mont_square_termination.v`	138 lines
	`multi_sub_termination.v`	123 lines
	`multi_lt_termination.v`	224 lines
	`multi_zero_termination.v`	71 lines
Semantics extraction (Sect. 5.2)	`bbs_encode_decode.v`	959 lines
Implementation step (Sect. 5.3)	`BBS_Asm_CryptoProof.v`	270 lines
Total		2374 lines

Table 4: Security proof for BBS in assembly

8. Related Work

Our work combines a framework for the formal verification of cryptographic assembly code [AM06] with a framework for the formal verification of security proofs [Now07].

For the formal verification of assembly code, one may think of alternative approaches, that may potentially alleviate the burden of interactive proof in Hoare logic. Proof-producing compilation (from a high-level language suitable for specification, down to assembly code) is such an approach. However, it is not clear that a proof-producing compiler (such as the one of [MSG09]) can be instrumented so as to yield code that is as efficient as hand-written assembly code (as typically found in cryptographic software), especially when the latter can take advantage of instructions tailored to the implementation of cryptography. Formal verification by refinement from a functional specification is sometimes considered as an approach that potentially leads to shorter proof scripts. However, application of this approach to cryptographic assembly code in [MG07] does not seem to reconcile short proof scripts with compact assembly code.

We think that our work shows that interactive proof in Hoare logic can be a viable approach when the framework for program verification is adequately instrumented. For example, the proof-carrying code framework of [SU07] turns out to be instrumental in dealing with assembly with jumps. Also, we find separation logic [Rey02] convenient and expressive to write and prove triples involving pointers; nevertheless, it should be noted that there exist alternative ways to conveniently express separation properties (e.g., directly in higher-order logic [MN05]). Our approach is based on a faithful model of assembly code with a compilation function that does not perform complex transformations; alternatively, one may want to experiment formal verification with a more abstract model by deferring burden on a more involved compilation function: such an approach may bring more automation (à la [FM04]) to formal verification.

As for the formal verification of security proofs, there exist several frameworks based on proof assistants [BBU08, BGZ09, Now07]. They all rely on games [Sho04] but differ on how they implement them. More precisely, games are shallow embedded in [Now07] (they are modeled as Coq functions) and deep embedded in [BBU08, BGZ09] (their syntax and semantics are encoded by inductive types). It turns out that the two frameworks ([AM06] and [Now07]) that we integrate here are a good fit for they are both based on shallow embeddings: 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 addition, our use-case directly relies on properties of arithmetic (including an encoding of the quadratic residuosity problem) an originality of [Now08].

9. Conclusion

We addressed the problem of formal verification of assembly code with security proofs. We proposed an approach that extends game-playing to integrate formal proofs of functional correctness with formal security proofs in a clear way, understandable by both cryptographers and implementers. Our proposal 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 security proof.

Future Work. The security 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 security proof of the *concrete security* of BBS.

Our certified implementation of BBS can be used as the source of pseudorandomness in the implementation of further cryptographic primitives: one can extract their semantics as a mathematical function and inject it into the appropriate standard definition of security (such as *semantic security* in the case of ElGamal). Our approach could also be extended to deal with nondeterminism and thus certify cryptographic primitives relying on sources of truly random bits. Another extension worth investigation is the support for probabilistic termination that one can find, e.g., in probabilistic decryption.

Acknowledgments. This work was partially supported by KAKENHI 21500046 and 21700048. The authors are grateful to the anonymous reviewers for their helpful comments.

References

[AM06] R. Affeldt, N. Marti, An Approach to Formal Verification of Arithmetic Functions in Assembly, in: M. Okada, I. Satoh (Eds.), Revised Selected Papers of the 11th Asian Computing Science Conference, Secure Software and Related Issues, ASIAN 2006, in: Lecture Notes in Computer Science, vol. 4435, Springer, 2008, pp. 346–360.

[ANY09] R. Affeldt, D. Nowak, K. Yamada, Certifying Assembly with Security Proofs: the Case of BBS, in: Electronic Communications of the EASST, vol. 23, 2009.

[ANY11] R. Affeldt, D. Nowak, K. Yamada, Certifying Assembly with Security Proofs: the Case of BBS, Coq development, `http://staff.aist.go.jp/reynald.affeldt/bbs`, last access: 2011/01/14.

[ATM07] R. Affeldt, M. Tanaka, N. Marti, Formal Proof of Provable Security by Game-Playing in a Proof Assistant, in: W. Susilo, J.K. Liu, Y. Mu (Eds.), Proceedings of the 1st International Conference on Provable Security, Provsec 2007, in: Lecture Notes in Computer Science, vol. 4784, Springer, pp. 151–168.

[BBU08] M. Backes, M. Berg, D. Unruh, A Formal Language for Cryptographic Pseudocode, in: I. Cervesato, H. Veith, A. Voronkov (Eds.), Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2008, in:

Lecture Notes in Computer Science, vol. 5330, Springer, pp. 353–376.

[BGZ09] G. Barthe, B. Grégoire, S. Zanella-Béguelin, Formal certification of code-based cryptographic proofs, in: Z. Shao, B.C. Pierce (Eds.), Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, ACM, pp. 90–101.

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

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

[Coq10] The Coq Development Team, The Coq Proof Assistant Reference Manual — Version 8.3, `http://coq.inria.fr/refman`, INRIA, 2010, last access: 2011/01/14.

[DK07] H. Delfs, H. Knebl, Introduction to Cryptography: Principles and Applications, Second Edition, Springer, 2007.

[FM04] J.C. Filliâtre, C. Marché, Multi-prover Verification of C Programs, in: J. Davies, W. Schulte, M. Barnett (Eds.), Proceedings of the 6th International Conference on Formal Engineering Methods, ICFEM 2004, in: Lecture Notes in Computer Science, vol. 3308, Springer, pp. 15–29.

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

[MAY06] N. Marti, R. Affeldt, A. Yonezawa, Formal Verification of the Heap Manager of an Operating System using Separation Logic, in: Z. Liu, J. He (Eds.), Proceedings of the 8th International Conference on Formal Engineering Methods, ICFEM 2006, in: Lecture Notes in Computer Science, vol. 4260, Springer, pp. 400–419.

[MN05] F. Mehta, T. Nipkow, Proving pointer programs in higher-order logic, Information and Computation 199 (1–2) (2005) 200–227.

[MOV01] A. Menezes, P.C. van Oorschot, S.A. Vanstone, Handbook of Applied Cryptography, fifth printing, CRC Press, 2001.

[Mips01] MIPS Technologies, MIPS32 4KS Processor Core Family Software User's Manual, 2001.

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

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

[MSG09] M.O. Myreen, K. Slind, M.J.C. Gordon, Extensible proof-producing compilation, in: O. de Moor, M.I. Schwartzbach (Eds.), Proceedings of the 18th International Conference on Compiler Construction, CC 2009, in: Lecture Notes in Computer Science, vol. 5501, Springer, pp. 2–16.

[Now07] D. Nowak, A framework for game-based security proofs, in: S. Qing, H. Imai, G. Wang (Eds.), Proceedings of the 9th International Conference on Information and Communications Security, ICICS 2007, in: Lecture Notes in Computer Science, vol. 4861, Springer, pp. 319–333.

[Now08] D. Nowak, On formal verification of arithmetic-based cryptographic primitives, in: P.J. Lee, J.H. Cheon (Eds.), Revised Selected Papers of the 11th International Conference on Information Security and Cryptology, ICISC 2008, in: Lecture Notes in Computer Science, vol. 5461, Springer, 2009, pp. 368-382.

[Rey02] J.C. Reynolds, Separation Logic: A Logic for Shared Mutable Data Structures, in: Proceedings of the 17th IEEE Symposium on Logic in Computer Science, LICS 2002, IEEE Computer Society, pp. 55–74.

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

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

[TA06] G. Tan, A.W. Appel, A Compositional Logic for Control Flow, in: E. Allen Emerson, K.S. Namjoshi (Eds.), Proceedings of 7th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2006, in: Lecture Notes in Computer Science, vol. 3855, Springer, pp. 80–94.

[Yam10] K. Yamada, Bug report and patch for SmartMIPS support in GDB, `http://article.gmane.org/gmane.comp.gdb.patches/55961`, 2010/03/09, last access: 2011/01/14.

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