Verification of Dependable Software using SPARK and Isabelle

Stefan Berghofer*

secunet Security Networks AG
Ammonstraße 74, 01067 Dresden, Germany

Abstract. We present a link between the interactive proof assistant Isabelle/HOL and the SPARK/Ada tool suite for the verification of high-integrity software. Using this link, we can tackle verification problems that are beyond reach of the proof tools currently available for SPARK. To demonstrate that our methodology is suitable for real-world applications, we show how it can be used to verify an efficient library for big numbers. This library is then used as a basis for an implementation of the RSA public-key encryption algorithm in SPARK/Ada.

1 Introduction

Software for security-critical applications, such as a data encryption algorithm in a virtual private network (VPN) gateway, needs to be particularly trustworthy. If the encryption algorithm does not work as specified, data transmitted over the network may be decrypted or manipulated by an adversary. Moreover, flaws in the implementation may also make the VPN gateway vulnerable to overflows, enabling an attacker to obtain access to the system, or cause the whole gateway to crash. If such a gateway is part of the VPN of a bank, implementation flaws can easily cause considerable financial damage. For that reason, there is a strong economic motivation to avoid bugs in software for such application areas.

Since software controls more and more areas of daily life, software bugs have received increasing attention. In 2006, a bug was introduced into the key generation tool of OpenSSL that was part of the Debian distribution. As a consequence of this bug, the random number generator for producing the keys no longer worked properly, making the generated keys easily predictable and therefore insecure [6]. This bug went unnoticed for about two years.

Although it is commonly accepted that the only way to make sure that software conforms to its specification is to *formally prove* its correctness, it was not until recently that verification tools have reached a sufficient level of maturity to be industrially applicable. A prominent example of such a tool is the SPARK system [2]. It is developed by Altran Praxis and is widely used in industry, notably in the area of avionics. SPARK is currently being used to develop the UK's next-generation air traffic control system *iFACTS*, and has already been

* Supported by Federal Office for Information Security (BSI) under grant 880

successfully applied to the verification of a biometric software system in the context of the *Tokeneer* project funded by the NSA [3]. The SPARK system analyzes programs written in a subset of the Ada language, and generates logical formulae that need to hold in order for the programs to be correct. Since it is undecidable in general whether a program meets its specification, not all of these generated formulae can be proved automatically. In this paper, we therefore present the HOL-SPARK verification environment that couples the SPARK system with the interactive proof assistant *Isabelle/HOL* [13].

SPARK imposes a number of restrictions on the programmer to ensure that programs are well-structured and thus more easily verifiable. Pointers and GOTOs are banned from SPARK programs, and for each SPARK procedure, the programmer must declare the intended direction of dataflow. This may sound cumbersome, but eventually leads to code of much higher quality. In standard programming languages, requirements on input parameters or promises about output parameters of procedures, also called *pre-* and *postconditions*, such as "`i` must be smaller than the length of the array `A`" or "`x` will always be greater than 1" are usually written as comments in the program, if at all. These comments are not automatically checked, and often they are wrong, for example when a programmer modified a piece of code but forgot to ensure that the comment still reflects the actual behaviour of the code. SPARK allows the programmer to write down pre- and postconditions of a procedure as logical formulae, and a link between these conditions and the code is provided by a formal correctness proof of the procedure, which makes it a lot easier to detect missing requirements. Moreover, the obligation to develop the code in parallel with its specification and correctness proof facilitates the production of code that immediately works as expected, without spending hours on testing and bug fixing. Having a formal correctness proof of a program also makes it easier for the programmer to ensure that changes do not break important properties of the code.

The rest of this paper is structured as follows. In §2, we give some background information about SPARK and our verification tool chain. In §3, we illustrate the use of our verification environment with a small example. As a larger application, we discuss the verification of a big number library in §4. A brief overview of related work is given in §5. Finally, §6 contains an evaluation of our approach and an outlook to possible future work.

2 Basic Concepts

2.1 SPARK

SPARK [2] is a subset of the Ada language that has been designed to allow verification of high-integrity software. It is missing certain features of Ada that can make programs difficult to verify, such as *access types*, *dynamic data structures*, and *recursion*. SPARK allows to prove absence of *runtime exceptions*, as well as *partial correctness* using pre- and postconditions. Loops can be annotated with *invariants*, and each procedure must have a *dataflow annotation*, specifying the

dependencies of the output parameters on the input parameters of the procedure. Since SPARK annotations are just written as comments, SPARK programs can be compiled by an ordinary Ada compiler such as GNAT. SPARK comes with a number of tools, notably the *Examiner* that, given a SPARK program as an input, performs a *dataflow analysis* and generates *verification conditions* (VCs) that must be proved in order for the program to be exception-free and partially correct. The VCs generated by the Examiner are formulae expressed in a language called FDL, which is first-order logic extended with arithmetic operators, arrays, records, and enumeration types. For example, the FDL expression

```
for_all(i: integer, ((i >= min) and (i <= max)) ->
  (element(a, [i]) = 0))
```

states that all elements of the array `a` with indices greater or equal to `min` and smaller or equal to `max` are 0. VCs are processed by another SPARK tool called the *Simplifier* that either completely solves VCs or transforms them into simpler, equivalent conditions. The latter VCs can then be processed using another tool called the *Proof Checker*. While the Simplifier tries to prove VCs in a completely automatic way, the Proof Checker requires user interaction, which enables it to prove formulae that are beyond the scope of the Simplifier. The steps that are required to manually prove a VC are recorded in a log file by the Proof Checker. Finally, this log file, together with the output of the other SPARK tools mentioned above, is read by a tool called POGS (**P**roof **O**bli**G**ation **S**ummariser) that produces a table mentioning for each VC the method by which it has been proved. In order to overcome the limitations of FDL and to express complex specifications, SPARK allows the user to declare so-called *proof functions*. The desired properties of such functions are described by postulating a set of rules that can be used by the Simplifier and Proof Checker [2, §11.7]. An obvious drawback of this approach is that incorrect rules can easily introduce inconsistencies.

2.2 HOL-SPARK

The HOL-SPARK verification environment, which is built on top of Isabelle's object logic HOL, is intended as an alternative to the SPARK Proof Checker, and improves on it in a number of ways. HOL-SPARK allows Isabelle to directly parse files generated by the Examiner and Simplifier, and provides a special proof command to conduct proofs of VCs, which can make use of the full power of Isabelle's rich collection of proof methods. Proofs can be conducted using Isabelle's graphical user interface, which makes it easy to navigate through larger proof scripts. Moreover, proof functions can be introduced in a *definitional* way, for example by using Isabelle's package for recursive functions, rather than by just stating their properties as axioms, which avoids introducing inconsistencies.

Figure 1 shows the integration of HOL-SPARK into the tool chain for the verification of SPARK programs. HOL-SPARK processes declarations (`*.fdl`) and rules (`*.rls`) produced by the Examiner, as well as simplified VCs (`*.siv`) produced by the SPARK Simplifier. Alternatively, the original unsimplified VCs (`*.vcg`)

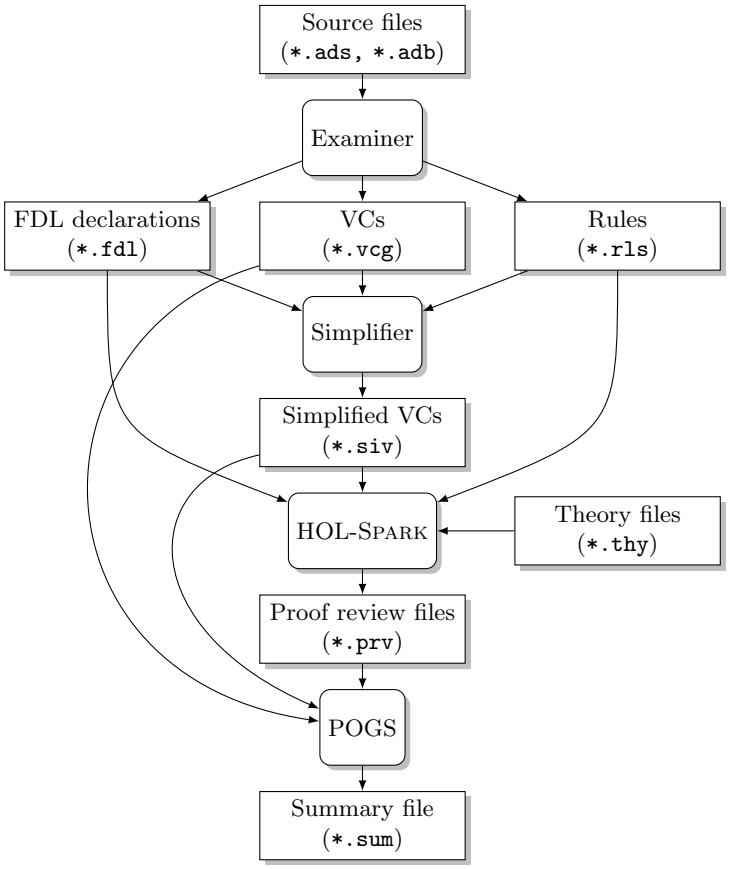

Fig. 1. SPARK program verification tool chain

produced by the Examiner can be used as well. Processing of the SPARK files is triggered by an Isabelle theory file (`*.thy`), which also contains the proofs for the VCs contained in the `*.siv` or `*.vcg` files. Once that all verification conditions have been successfully proved, Isabelle generates a proof review file (`*.prv`) notifying the POGS tool of the VCs that have been discharged.

3 Verifying an Example Program

In this section, we explain the usage of the SPARK verification environment by proving the correctness of an example program for computing the *greatest common divisor* of two natural numbers shown in Fig. 2, which has been taken from the book about SPARK by Barnes [2, §11.6]. In order to specify that the SPARK procedure `G_C_D` behaves like its mathematical counterpart, Barnes introduces a proof function `Gcd` in the package specification.

```
package Greatest_Common_Divisor
is
   --# function Gcd (A, B : Natural) return Natural;

   procedure G_C_D (M, N : in Natural; G : out Natural);
     --# derives G from M, N;
     --# post G = Gcd (M, N);

end Greatest_Common_Divisor;

package body Greatest_Common_Divisor
is
   procedure G_C_D (M, N : in Natural; G : out Natural)
   is
      C, D, R : Natural;
   begin
      C := M; D := N;
      while D /= 0
        --# assert Gcd (C, D) = Gcd (M, N);
      loop
         R := C mod D;
         C := D; D := R;
      end loop;
      G := C;
   end G_C_D;

end Greatest_Common_Divisor;
```

Fig. 2. SPARK program for computing the greatest common divisor

3.1 Importing SPARK VCs into Isabelle

Invoking the Examiner and Simplifier on this program yields a file `g_c_d.siv` containing the simplified VCs, as well as files `g_c_d.fdl` and `g_c_d.rls`, containing FDL declarations and rules, respectively. For `G_C_D` the Examiner generates nine VCs, seven of which are proved automatically by the Simplifier. We now show how to prove the remaining two VCs interactively using HOL-SPARK. For this purpose, we create a *theory* `Greatest_Common_Divisor`, which is shown in Fig. 3. Each proof function occurring in the specification of a SPARK program must be linked with a corresponding Isabelle function. This is accomplished by the command **spark_proof_functions**, which expects a list of equations *name* `=` *term*, where *name* is the name of the proof function and *term* is the corresponding Isabelle term. In the case of `gcd`, both the SPARK proof function and its Isabelle counterpart happen to have the same name. Isabelle checks that the type of the term linked with a proof function matches the type of the function declared in the `*.fdl` file. We now instruct Isabelle to open a new verification environment and load a set of VCs. This is done using the command **spark_open**, which

```
theory Greatest_Common_Divisor
imports SPARK GCD
begin

spark_proof_functions
  gcd = "gcd :: int ⇒ int ⇒ int"

spark_open "out/greatest_common_divisor/g_c_d.siv"

spark_vc procedure_g_c_d_4
  using `0 < d` `gcd c d = gcd m n`
  by (simp add: gcd_non_0_int)

spark_vc procedure_g_c_d_9
  using `0 ≤ c` `gcd c 0 = gcd m n`
  by simp

spark_end

end
```

Fig. 3. Correctness proof for the greatest common divisor program

must be given the name of a `*.siv` or `*.vcg` file as an argument. Behind the scenes, Isabelle parses this file and the corresponding `*.fdl` and `*.rls` files, and converts the VCs to Isabelle terms.

3.2 Proving the VCs

The two open VCs are `procedure_g_c_d_4` and `procedure_g_c_d_9`, both of which contain the `gcd` proof function that the Simplifier does not know anything about. The proof of a particular VC can be started with the **spark_vc** command. The VC `procedure_g_c_d_4` requires us to prove that the `gcd` of `d` and the remainder of `c` and `d` is equal to the `gcd` of the original input values `m` and `n`, which is the *invariant* of the procedure. This is a consequence of the following theorem

```
0 < y ⟹ gcd x y = gcd y (x mod y)
```

The VC `procedure_g_c_d_9` says that if the loop invariant holds when we exit the loop, which means that `d = 0`, then the postcondition of the procedure will hold as well. To prove this, we observe that `gcd c 0 = c` for non-negative `c`. This concludes the proofs of the open VCs, and hence the SPARK verification environment can be closed using the command **spark_end**. This command checks that all VCs have been proved and issues an error message otherwise. Moreover, Isabelle checks that there is no open SPARK verification environment when the final **end** command of a theory is encountered.

4 A verified big number library

We will now apply the HOL-Spark environment to the verification of a library for big numbers. Libraries of this kind form an indispensable basis of algorithms for public key cryptography such as RSA or elliptic curves, as implemented in libraries like OpenSSL. Since cryptographic algorithms involve numbers of considerable size, for example 256 bytes in the case of RSA, or 40 bytes in the case of elliptic curves, it is important for arithmetic operations to be performed as efficiently as possible.

4.1 Introduction to modular multiplication

An operation that is central to many cryptographic algorithms is the computation of $x \cdot y \bmod m$, which is called *modular multiplication*. An obvious way of implementing this operation is to apply the standard multiplication algorithm, followed by division. Since division is one of the most complex operations on big numbers, this approach would not only be very difficult to implement and verify, but also computationally expensive. Therefore, big number libraries often use a technique called *Montgomery multiplication* [10, §14.3.2]. We can think of a big number x as an array of words $x_0, \ldots, x_{n-1}$, where $0 \leq x_i$ and $x_i < b$, and

$$x = \sum_{0 \leq i < n} b^i \cdot x_i$$

In implementations, b will usually be a power of 2. For two big numbers x and y, Montgomery multiplication (denoted by $x \otimes y$) yields

$$x \otimes y = x \cdot y \cdot R^{-1} \bmod m$$

where $R = b^n$, and R^{-1} denotes the multiplicative inverse of R modulo m. Now, in order to compute the product of two numbers x and y modulo m, we first compute the *residues* $\widetilde{x}$ and $\widetilde{y}$ of these numbers, where $\widetilde{x} = x \cdot R \bmod m$ and $\widetilde{y}$ likewise. A residue $\widetilde{x}$ can be computed by a Montgomery multiplication of x with $R^2 \bmod m$, since

$$x \otimes (R^2 \bmod m) = x \cdot R^2 \cdot R^{-1} \bmod m = x \cdot R \bmod m$$

We then have that

$$\widetilde{x} \otimes \widetilde{y} = x \cdot R \cdot y \cdot R \cdot R^{-1} \bmod m = x \cdot y \cdot R \bmod m = \widetilde{x \cdot y}$$

The desired result of the modular multiplication can be obtained by performing a Montgomery multiplication of $\widetilde{x \cdot y}$ with 1, since

$$\widetilde{x \cdot y} \otimes 1 = x \cdot y \cdot R \cdot 1 \cdot R^{-1} \bmod m = x \cdot y \bmod m$$

Before we come to the implementation and verification of Montgomery multiplication, we try to give an intuitive explanation of how the algorithm works. Our

$a \leftarrow 0$
for $i = n - 1$ **downto** 0 **do**
  $a \leftarrow a \cdot b + x_i \cdot y$
end for

	$0 \cdot 10 =$
	$0 +$
$4 \cdot 789$	$3156 =$
	$3156 \cdot 10 =$
	$31560 +$
$5 \cdot 789$	$3945 =$
	$35505 \cdot 10 =$
	$355050 +$
$6 \cdot 789$	$4734 =$
	359784

$a \leftarrow 0$
for $i = 0$ **to** $n - 1$ **do**
  $a \leftarrow (a + x_i \cdot y)/b$
end for

	0	$+$
$6 \cdot 789$	4734	$=$
	4734	$/\ 10 =$
	473.4	$+$
$5 \cdot 789$	3945	$=$
	4418.4	$/\ 10 =$
	441.84	$+$
$4 \cdot 789$	3156	$=$
	3597.84	$/\ 10 =$
	359.784	

Fig. 4. Two variants of multiplication

exposition is inspired by a note due to Kochanski [9]. As a running example, we take $b = 10$ and assume we would like to multiply 456 with 789. Fig. 4 shows two multiplication algorithms in pseudocode notation, and the tables below the algorithms illustrate the computation steps performed by them. The algorithm on the left is the usual "school multiplication": the multiplier x is processed from left to right, i.e. starting with the most significant digit, and the accumulator a is shifted to the left, i.e. multiplied with 10 in each step. In contrast, the algorithm on the right processes the multiplier from right to left, i.e. starting with the least significant digit, and shifts the accumulator to the right, i.e. divides it by 10. Consequently, the algorithm on the right computes $x \cdot y \cdot R^{-1}$ instead of $x \cdot y$. We now explain how the algorithm on the right can be modified to perform modular multiplication. It might seem that the algorithm requires computations involving floating point numbers, since $a + x_i \cdot y$ is not necessarily divisible by b. However, when working modulo m, this can easily be fixed by adding a suitable multiple of m to $a + x_i \cdot y$, which does not change the result modulo m. The factor by which we have to multiply m is $u = (a_0 + x_i \cdot y_0) \cdot m' \bmod b$, where $m' = -m_0^{-1} \bmod b$ is the additive inverse of the multiplicative inverse of m_0 modulo b, i.e. $(1 + m' \cdot m_0) \bmod b = 0$ and $0 \leq m' < b$. The inverse only exists if m_0 and b are *coprime*, i.e. $gcd(m_0, b) = 1$, which is the case in practical applications, since b will usually be a power of 2 and m will be a large prime number. Note that in order to compute u, we only have to consider the least significant words a_0, y_0 and m_0 of the numbers a, y and m, respectively. It is easy to see that $a + x_i \cdot y + u \cdot m$ is divisible by b, since

$$(a + x_i \cdot y + u \cdot m) \bmod b = (a_0 + x_i \cdot y_0 + (a_0 + x_i \cdot y_0) \cdot m' \cdot m_0) \bmod b =$$
$$(a_0 + x_i \cdot y_0) \cdot (1 + m' \cdot m_0) \bmod b = 0$$

Fig. 5 shows the pseudocode for the Montgomery multiplication algorithm, which employs the ideas described above. As for the other algorithms, we also include

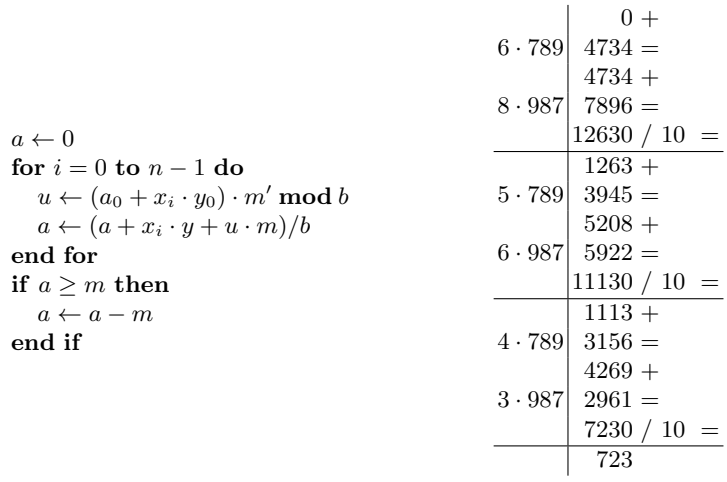

Fig. 5. Montgomery multiplication algorithm

a table illustrating the computation. We again multiply the numbers 456 and 789, and use 987 as a modulus. Note that $m' = 7$, since $(1 + 7 \cdot 7) \bmod 10 = 0$. The result of the multiplication is easily seen to be correct, since

$$723 \cdot 1000 \bmod 987 = 516 = 456 \cdot 789 \bmod 987$$

After termination of the loop, it may be necessary to subtract m from a, since a may not be smaller than m, although it will always be smaller than $2 \cdot m - 1$.

4.2 Overview of the big number library

In this section, we give an overview of the big number library and its interface. We have chosen to represent big numbers as *unconstrained arrays* of 64-bit words, where the array indices can range over the natural numbers. All procedures in the big number library operate on segments of unconstrained arrays that are selected by specifying the first and last index of the segment. In situations where a procedure operates on several segments, all of which must have the same length, the last index is usually omitted. The prelude of the `Bignum` library containing the basic declarations is shown in Fig. 6. The big number library provides the following operations:

- Basic big number operations: doubling, subtracting, and comparing
- Precomputation of the values $R^2 \bmod m$ and $-m_0^{-1} \bmod b$
- Montgomery multiplication
- Exponentiation using Montgomery multiplication

The value $R^2 \bmod m = \left(\left(2^k\right)^n\right)^2 \bmod m = 2^{2 \cdot k \cdot n} \bmod m$ can be computed by initializing an accumulator with 1 and applying the doubling operation to it $2 \cdot k \cdot n$

```
package Bignum
is
   Word_Size : constant := 64;
   Base : constant := 2 ** Word_Size;
   type Word is mod Base;
   type Big_Int is array (Natural range <>) of Word;

   --# function Num_Of_Big_Int (A: Big_Int; K, I: Natural)
   --#   return Universal_Integer;

   --# function Num_Of_Boolean (B: Boolean)
   --#   return Universal_Integer;

   --# function Inverse (M, X: Universal_Integer)
   --#   return Universal_Integer;
                                        ...
end Bignum;
```

Fig. 6. Prelude of the big number library

times. After each doubling step, we check whether a carry bit was produced or the resulting number is greater or equal to m, in which case we have to subtract m from the current value of the accumulator. The value $-m_0^{-1}$ **mod** b can be computed by a variant of Euclid's algorithm shown in §3.

Since the specification of the big number operations will make use of constructs that cannot be easily expressed with SPARK's annotation language, we have to introduce a number of proof functions. First of all, we need a function that abstracts a big number to a number in the mathematical sense. This function, which is called `Num_Of_Big_Int`, takes an array `A`, together with the first index `K` and the length `I` of the segment representing the big number, and returns a result of type `Universal_Integer`. The Isabelle counterpart of this function is

```
num_of_big_int :: (int ⇒ int) ⇒ int ⇒ int ⇒ int
num_of_big_int A k i = (∑ j = 0..<i. Base^j * A (k + j))
```

An array with elements of type τ is represented by the function type `int` $\Rightarrow \tau$ in Isabelle. Function `num_of_big_int` enjoys the following summation property

```
num_of_big_int A k (i + j) =
num_of_big_int A k i + Base^i * num_of_big_int A (k + i) j
```

It is important to note that it would not have been adequate to choose `Integer` instead of `Universal_Integer` as a result type, since the former corresponds to *machine integers* limited to a fixed size, whereas the latter corresponds to the *mathematical* ones. When dealing with operations returning *carry bits*, it is often useful to have a function for converting boolean values to numbers, where

```
procedure Mont_Mult
  (A : out Big_Int; A_First : in Natural; A_Last : in Natural;
   X : in Big_Int; X_First : in Natural;
   Y : in Big_Int; Y_First : in Natural;
   M : in Big_Int; M_First : in Natural;
   M_Inv : in Word);
--# derives
--#   A from
--#   A_First, A_Last, X, X_First, Y, Y_First, M, M_First, M_Inv;
--# pre
--#   A_First in A'Range and A_Last in A'Range and
--#   A_First < A_Last and
--#   X_First in X'Range and
--#   X_First + (A_Last - A_First) in X'Range and
--#   ...
--#   Num_Of_Big_Int (Y, Y_First, A_Last - A_First + 1) <
--#   Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) and
--#   1 < Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) and
--#   1 + M_Inv * M (M_First) = 0;
--# post
--#   Num_Of_Big_Int (A, A_First, A_Last - A_First + 1) =
--#   (Num_Of_Big_Int (X, X_First, A_Last - A_First + 1) *
--#    Num_Of_Big_Int (Y, Y_First, A_Last - A_First + 1) *
--#    Inverse (Num_Of_Big_Int (M, M_First, A_Last - A_First + 1),
--#      Base) ** (A_Last - A_First + 1)) mod
--#   Num_Of_Big_Int (M, M_First, A_Last - A_First + 1);
```

Fig. 7. Specification of Montgomery multiplication

`False` and `True` are converted to 0 and 1, respectively. This is accomplished by the proof function `Num_Of_Boolean`. Finally, for writing down the specification of Montgomery multiplication, we also need the proof function `Inverse` denoting the multiplicative inverse of `X` modulo `M`. It corresponds to the Isabelle function `minv::int ⇒ int ⇒ int`, which has the following central property

```
coprime x m ⟹ 0 < x ⟹ 1 < m ⟹ x * minv m x mod m = 1
```

Moreover, if `n'` is the multiplicative inverse of `n` modulo `m`, multiplying `k` by `n'` is equivalent modulo `m` to dividing `k` by `n`, provided that `k` is divisible by `n`:

```
n * n' mod m = 1 ⟹ k mod n = 0 ⟹ k div n mod m = k * n' mod m
```

This property does not hold if `k mod n ≠ 0`. For example, `5 * 13 mod 16 = 1` and `10 * 13 mod 16 = 2 = 10 div 5`, but `9 * 13 mod 16 = 5 ≠ 1 = 9 div 5`.

4.3 Montgomery multiplication

The central operation in the big number library is Montgomery multiplication, whose specification is shown in Fig. 7. It multiplies `X` with `Y` and stores the result in `A`. The precondition requires the second factor `Y` to be smaller than the modulus `M`. Due to the construction of the algorithm, the first factor `X` is not required to be smaller than `M` in order for the result to be correct. For technical reasons, `A_Last` must be greater than `A_First`, i.e. the length of the big number must be at least 2. This is not a serious restriction, since big numbers of length 1 would be rather pointless. Moreover, the modulus is required to be greater than 1. The precondition `1 + M_Inv * M (M_First) = 0` states that `M_Inv` must be the additive inverse of the multiplicative inverse modulo b of the least significant word of the modulus. The postcondition essentially states that $a = x \cdot y \cdot (b^{-1})^n \bmod m$, where n is the length of the big numbers involved, and a, x, y, m are the numbers represented by the arrays `A`, `X`, `Y`, `M`, respectively.

We are now ready to describe the implementation of Montgomery multiplication, which is shown in Fig. 8. Recall that in each step of the Montgomery multiplication algorithm outlined in §4.1, we have to compute $(a + x_i \cdot y + u \cdot m)/b$, where x_i and u are words, and a, y and m are big numbers. In our code for computing this value, we use an optimization technique suggested by Myreen [12, §3.2], which he used for the verification of an ARM machine code implementation of Montgomery multiplication in HOL4. The idea is to perform the two multiplications of a word with a big number, as well as the two addition operations in one single loop. The computation will be done in-place, meaning that the old value of a will be overwritten with the new value. Moreover, since $a + x_i \cdot y + u \cdot m$ is divisible by b, we also shift the array containing the result by one word to the left while performing the computation, which corresponds to a division by b. This is accomplished by the procedure `Add_Mult_Mult` with postcondition

```
Num_Of_Big_Int (A~, A_First + 1, A_Last - A_First + 1) +
Num_Of_Big_int (Y, Y_First, A_Last - A_First + 1) * XI +
Num_Of_Big_int (M, M_First, A_Last - A_First + 1) * U +
Carry1~ + Base * Carry2~ =
Num_Of_Big_Int (A, A_First, A_Last - A_First + 1) +
Base ** (A_Last - A_First + 1) * (Carry1 + Base * Carry2)
```

The array representing $(a + x_i \cdot y + u \cdot m)/b$ needs to be one word longer than the length of y and m, although the final result of Montgomery multiplication will have the same length as the input numbers. We therefore store the most significant word of a in a separate variable `A_MSW` that is discarded at the end of the computation. To simplify the implementation of the computation described above, we first implement an auxiliary procedure `Single_Add_Mult_Mult` for computing $a_j + x_i \cdot y_j + u \cdot m_j$, where all the operands involved are words. Procedure `Add_Mult_Mult` just iteratively applies this auxiliary procedure to the elements of the big numbers involved.

The **assert** annotation after the **for** command in Fig. 8 specifies the loop invariant, which is

```
procedure Mont_Mult
  ...
is
   Carry : Boolean;
   Carry1, Carry2, A_MSW, XI, U : Word;
begin
   Initialize (A, A_First, A_Last); A_MSW := 0;

   for I in Natural range A_First .. A_Last
     --# assert ...
   loop
      Carry1 := 0; Carry2 := 0;
      XI := X (X_First + (I - A_First));
      U := (A (A_First) + XI * Y (Y_First)) * M_Inv;
      Single_Add_Mult_Mult
        (A (A_First), XI, Y (Y_First),
         M (M_First), U, Carry1, Carry2);
      Add_Mult_Mult
        (A, A_First, A_Last - 1,
         Y, Y_First + 1, M, M_First + 1,
         XI, U, Carry1, Carry2);
      A (A_Last) := A_MSW + Carry1;
      A_MSW := Carry2 + Word_Of_Boolean (A (A_Last) < Carry1);
   end loop;

   if A_MSW /= 0 or else
     not Less (A, A_First, A_Last, M, M_First) then
      Sub_Inplace (A, A_First, A_Last, M, M_First, Carry);
   end if;
end Mont_Mult;
```

Fig. 8. Implementation of Montgomery multiplication

```
(Num_Of_Big_Int (A, A_First, A_Last - A_First + 1) +
 Base ** (A_Last - A_First + 1) * A_MSW) mod
Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) =
(Num_Of_Big_Int (X, X_First, I - A_First) *
 Num_Of_Big_Int (Y, Y_First, A_Last - A_First + 1) *
 Inverse (Num_Of_Big_Int (M, M_First, A_Last - A_First + 1),
   Base) ** (I - A_First)) mod
Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) and
Num_Of_Big_Int (A, A_First, A_Last - A_First + 1) +
Base ** (A_Last - A_First + 1) * A_MSW <
2 * Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) - 1
```

Using a more compact mathematical notation, this invariant can be written as

$$a \bmod m = (x|_j \cdot y \cdot b^{-j}) \bmod m \wedge a < 2 \cdot m - 1$$

where $x|_j$ denotes the number represented by the segment of the array `X` of length $j = \texttt{I} - \texttt{A\_First}$ starting at index `X_First`. The result a computed by the loop can be greater or equal to the modulus, in which case we have to subtract the modulus `M` in order to get the desired result. If $\texttt{A\_MSW} \neq 0$, this obviously means that $m < a$. If $\texttt{A\_MSW} = 0$, we have to check whether $m \leq a$ Since $a < 2 \cdot m - 1$, it suffices to subtract the modulus at most once [10, §14.3.2].

5 Related Work

The design of HOL-Spark is heavily inspired by the HOL-Boogie environment by Böhme et al. [4] that links Isabelle with Microsoft's Verifying C Compiler (VCC) [5]. The *Victor* tool by Jackson [8], which is distributed with the latest Spark release, uses a different approach. Victor is a command-line tool that can parse files produced by the Spark tools, and can transform them into a variety of formats, notably input files for SMT-solvers. Victor has recently been extended to produce Isabelle theory files as well. The drawback of using Victor in connection with Isabelle is that theory files have to be regenerated whenever there is a change in the files produced by Spark. This can happen quite frequently in the development phase, for example when the user notices that some loop invariant has to be strengthened, or the code has to be restructured in order to simplify verification. The Frama-C system and its *Jessie* plugin [11] for the verification of C code can generate VCs for a number of automatic and interactive provers, including Coq and Isabelle.

A similar big number library written in a C-like language has been proved correct in Isabelle/HOL by Fischer [7] using a verification environment due to Schirmer [14]. This library also includes division, but no Montgomery multiplication. Due to the use of linked lists with pointers instead of arrays, Fischer's formalization is a bit more complicated than ours. Apart from Myreen's work mentioned above, an implementation of Montgomery multiplication in MIPS assembly has been formalized using Coq by Affeldt and Marti [1].

6 Conclusion

We have developed a verification environment for Spark, which is already part of the Isabelle 2011 release, and have applied it to the verification of a big number library. Our implementation of RSA based on this library reaches about 40% of the speed of OpenSSL when compiled with the `-O3` option on a 64-bit platform. This is quite acceptable, given that OpenSSL uses highly-optimized and hand-written assembly code. A further performance gain could be achieved by using a sliding window exponentiation algorithm instead of the simpler square-and-multiply technique. The library has 743 LOCs, 316 of which (i.e. 43%) are Spark annotations. The length of the Isabelle files containing correctness proofs of all procedures in the library, as well as necessary background theory, is 1753

lines, of which 391 lines are taken up by the correctness proof for Montgomery multiplication. Development of the library, including proofs, took about three weeks. In the future, we plan to use the library as a basis for an implementation of elliptic curve cryptography. A more long-term goal is to embed the SPARK semantics into Isabelle, to further increase the trustworthiness of VC generation.

Acknowledgement I would like to thank Magnus Myreen for sharing the HOL4 proof scripts of his formalization of Montgomery multiplication with me. Sascha Böhme, Robert Dorn and Alexander Senier commented on draft versions of this paper and helped with optimizations and performance measurements.

References

1. R. Affeldt and N. Marti. An approach to formal verification of arithmetic functions in assembly. In M. Okada and I. Satoh, editors, *11th Annual Asian Computing Science Conf. 2006*, volume 4435 of *LNCS*, pages 346–360. Springer, 2008.
2. J. Barnes. *The* SPARK *Approach to Safety and Security.* Addison-Wesley, 2006.
3. J. Barnes, R. Chapman, R. Johnson, J. Widmaier, D. Cooper, and B. Everett. Engineering the tokeneer enclave protection software. In A. Hall and J. Wing, editors, *1st International Symposium on Secure Software Engineering.* IEEE, 2006.
4. S. Böhme, M. Moskal, W. Schulte, and B. Wolff. HOL-Boogie — An interactive prover-backend for the Verifying C Compiler. *Journal of Automated Reasoning*, 44(1–2):111–144, Feb. 2010.
5. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
6. Debian Security Advisory. DSA-1571-1 OpenSSL – predictable random number generator. Available online at http://www.debian.org/security/2008/dsa-1571.
7. S. Fischer. Formal verification of a big integer library written in C0. Master's thesis, Saarland University, 2006.
8. P. B. Jackson and G. O. Passmore. Proving SPARK Verification Conditions with SMT solvers, 2009.
9. M. Kochanski. Montgomery multiplication: a surreal technique. Available online at http://www.nugae.com/encryption/fap4/montgomery.htm.
10. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, October 1996.
11. Y. Moy and C. Marché. Jessie plugin tutorial. Technical report, INRIA, 2010. http://frama-c.com/jessie.html.
12. M. O. Myreen and M. J. C. Gordon. Verification of machine code implementations of arithmetic functions for cryptography. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings.* Dept. of Computer Science, University of Kaiserslautern, August 2007. Tech. report 364/07.
13. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS.* Springer-Verlag, 2002.
14. N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452, pages 398–414, 2005.

A RSA Encryption / Decryption

The implementation of the RSA encryption and decryption algorithm is shown in Fig. 9. The procedure `Crypt` computes $c = m^e \bmod n$, where c, m, e and n are the numbers represented by the arrays `C`, `M`, `E` and `N`, respectively. When used for encryption, c is the *ciphertext*, m the *plaintext message*, e the *public exponent*, and n the *modulus*, where n is the product of two prime numbers p and q, and $e \cdot d \bmod ((p-1) \cdot (q-1)) = 1$. The same procedure can be used to compute the plaintext from an encrypted message, i.e. $m = c^d \bmod n$. Before calling the Montgomery exponentiation algorithm explained in Appendix B, the procedure precomputes the values $R^2 \bmod n$ and $-n_0^{-1} \bmod b$. Since the exponentiation algorithm requires several auxiliary arrays for storing intermediate results of the computation, we define an array type of fixed length, which will be used for the message `M`, the modulus `N` and the ciphertext `C`:

```
subtype Mod_Range is Natural range 0 .. 63;
subtype Mod_Type is Bignum.Big_Int (Mod_Range);
```

This allows the `Crypt` function to allocate memory for the auxiliary arrays, rather than requiring the caller of `Crypt` to pass suitable arrays as arguments. We have set the length of `Mod_Type` to 64, meaning that it can contain values with $64 \cdot 64 = 4096$ bits, which is sufficient for most practical applications. However, the algorithm and its correctness proof would work equally well for different lengths of `Mod_Type`. Note that the length of the exponent `E` is still unconstrained and need not be the same as the length of the modulus. Indeed, it is quite common to choose public and private exponents that have a different length.

B Exponentiation

The implementation of exponentiation using Montgomery multiplication is shown in Fig. 10. This procedure computes the result $a = x^e \bmod m$, where a, x, e and m are the numbers represented by the arrays `A`, `X`, `E` and `M`, respectively. The algorithm needs a number of auxiliary variables to store intermediate values. These intermediate values are big numbers whose size is not known at compile time, but depends on the size of the unconstrained arrays passed as arguments to the procedure. Since SPARK does not allow the dynamic allocation of memory for data structures, these auxiliary variables need to be created by the caller, and passed to the procedure as arguments, too. This is why `Mont_Exp` has the extra arguments `Aux1`, `Aux2`, and `Aux3`. The parameter `RR` must contain the big number $R^2 \bmod m$, and $1 + \texttt{M\_Inv} \cdot m_0 \bmod b = 0$. We start by initializing `Aux1` with the big number 1. The variable `Aux3`, which we use as an accumulator for computing the result, is set to $\widetilde{1} = R \bmod m$ using `Mont_Mult` (see §4.1). Moreover, we store $\widetilde{x}$ in `Aux2`. The algorithm uses the square-and-multiply approach. It processes the exponent from the most significant bit to the least significant bit. In each iteration `Aux3` is squared, and the result stored in `A`. If the current

```
procedure Crypt
  (E : in Bignum.Big_Int;
   N : in Mod_Type;
   M : in Mod_Type;
   C : out Mod_Type)
is
   Aux1, Aux2, Aux3, RR : Mod_Type;
   N_Inv : Types.Word32;
begin
   Bignum.Size_Square_Mod
     (N, N'First, N'Last, RR, RR'First);

   N_Inv := Bignum.Word_Inverse (N (N'First));

   Bignum.Mont_Exp
     (C, C'First, C'Last,
      M, M'First,
      E, E'First, E'Last,
      N, N'First,
      Aux1, Aux1'First,
      Aux2, Aux2'First,
      Aux3, Aux3'First,
      RR, RR'First,
      N_Inv);
end Crypt;
```

Fig. 9. Implementation of RSA algorithm

bit of the exponent is set, `A` is multiplied with `Aux2` (containing $\widetilde{x}$), and the result is stored in `Aux3` again, otherwise `A` is just copied back to `Aux3`. The invariant of the inner loop is

```
Num_Of_Big_Int (Aux1, Aux1_First, A_Last - A_First + 1) = 1 and
Num_Of_Big_Int (Aux2, Aux2_First, A_Last - A_First + 1) =
Num_Of_Big_Int (X, X_First, A_Last - A_First + 1) *
Base ** (A_Last - A_First + 1) mod
Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) and
Num_Of_Big_Int (Aux3, Aux3_First, A_Last - A_First + 1) =
Num_Of_Big_Int (X, X_First, A_Last - A_First + 1) **
(Num_Of_Big_Int (E, I + 1, E_Last - I) * 2 ** (Word_Size - 1 - J) +
 Universal_Integer (E (I)) / 2 ** (J + 1)) *
Base ** (A_Last - A_First + 1) mod
Num_Of_Big_Int (M, M_First, A_Last - A_First + 1)
```

After termination of the loop, `Aux3` is converted from "Montgomery format" to the "normal format" again by Montgomery-multiplying it with 1 and storing the result in `A`.

```
procedure Mont_Exp
  (A : out Big_Int; A_First : in Natural; A_Last : in Natural;
   X : in Big_Int; X_First : in Natural;
   E : in Big_Int; E_First : in Natural; E_Last : in Natural;
   M : in Big_Int; M_First : in Natural;
   Aux1 : out Big_Int; Aux1_First : in Natural;
   ...
   RR : in Big_Int; RR_First : in Natural;
   M_Inv : in Word)
is
begin
   Initialize (Aux1, Aux1_First, Aux1_First + (A_Last - A_First));
   Aux1 (Aux1_First) := 1;

   Mont_Mult
     (Aux3, Aux3_First, Aux3_First + (A_Last - A_First),
      RR, RR_First, Aux1, Aux1_First, M, M_First, M_Inv);

   Mont_Mult
     (Aux2, Aux2_First, Aux2_First + (A_Last - A_First),
      X, X_First, RR, RR_First, M, M_First, M_Inv);

   for I in reverse Natural range E_First .. E_Last
   loop
      for J in reverse Natural range 0 .. Word_Size - 1
        --# assert ...
      loop
         Mont_Mult
           (A, A_First, A_Last,
            Aux3, Aux3_First, Aux3, Aux3_First,
            M, M_First, M_Inv);

         if (E (I) and 2 ** J) /= 0 then
            Mont_Mult
              (Aux3, Aux3_First, Aux3_First + (A_Last - A_First),
               A, A_First, Aux2, Aux2_First,
               M, M_First, M_Inv);
         else
            Copy (A, A_First, A_Last, Aux3, Aux3_First);
         end if;
      end loop;
   end loop;

   Mont_Mult
     (A, A_First, A_Last,
      Aux3, Aux3_First, Aux1, Aux1_First, M, M_First, M_Inv);
end Mont_Exp;
```

Fig. 10. Implementation of exponentiation