Verifying Cryptographic Software Correctness with Respect to Reference Implementations[⋆]

José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira

CCTC / Departamento de Informática
Universidade do Minho
Campus de Gualtar, 4710-Braga, Portugal
{jba,mbb,jsp,barbarasv}@di.uminho.pt

Abstract. This paper presents techniques developed to check program equivalences in the context of cryptographic software development, where specifications are typically reference implementations. The techniques allow for the integration of interactive proof techniques (required given the difficulty and generality of the results sought) in a verification infrastructure that is capable of discharging many verification conditions automatically. To this end, the difficult results in the verification process (to be proved interactively) are isolated as a set of lemmas. The fundamental notion of natural invariant is used to link the specification level and the interactive proof construction process.

1 Introduction

Software implementations of cryptographic algorithms and protocols are at the core of security functionality in many IT products. However, the development of this class of software products is understudied as a domain-specific niche in software engineering.

The development of cryptographic software is clearly distinct from other areas of software engineering due to a combination of factors. First of all, cryptography is an inherently inter-disciplinary subject. The design and implementation of cryptographic software draws on skills from mathematics, computer science and electrical engineering. The assumption that such a rich body of research can be absorbed and applied without error is tenuous for even the most expert software engineer. Secondly, security is notoriously difficult to sell as a feature in software products, even when clear risks such as identity theft and fraud are evident. An important implication of this fact is that security needs to be as close to invisible as possible in terms of computational and communication load. As a result, it is critical that cryptographic software is optimised aggressively, without altering the security semantics. Finally, typical software engineers develop systems focussed on desktop class processors within computers in our offices and homes. The special case of cryptographic software is implemented on a

[⋆] This work was partially supported by the European Union under the FP7-STREP project CACE (Project Number 216499), and by the FCT-funded RESCUE project (PTDC/EIA/65862/2006).

much wider range of devices, from embedded processors with very limited computational power, memory and autonomy, to high-end servers, which demand high-performance and low-latency. Not only must the cryptographic software engineers understand each platform and the related security requirements, they must also optimise each algorithm with respect to each platform since each will have vastly different performance characteristics.

CACE (Computer Aided Cryptography Engineering [5]) is an European Project that targets the lack of support currently offered to cryptographic software engineers. The central objective of the project is the development of a tool-box of domain-specific languages, compilers and libraries, that supports the production of high quality cryptographic software. The aim is that specific components within the tool-box will address particular software development problems and processes; and combined use of the constituent tools is enabled by designed integration between their interfaces. The project started in 2008 and will run for three years.

This paper stems from CACE - Work Package 5, which aims to add formal methods technology to the tool-box, as a means to increase the degree of assurance than can be provided by the development process. We describe promising early results obtained during our exploration of existing verification techniques and tools used to construct high-assurance software implementations for other domains. Specifically, we present our achievements in using an off-the-shelf verification tool to reason about the functional correctness of a `C` implementation of the `RC4` encryption scheme that is included in the well-known open-source library `openSSL` [15].

Contribution. The main contribution of this paper is to report on the application of the off-the-shelf `Frama-c` verification platform to verifying correctness of a real-world example of a cryptographic software implementation: the widely used `C` implementation of the `RC4` stream cipher available in the `openSSL` library. We focus on functional correctness, which is a critical use-case for verification tools when applied to cryptographic software. The (conceptual) specifications of cryptographic schemes are very often presented as pseudo-code algorithms, which may be easy to transcribe into a high-level programming language. However, given that cryptographic implementations are typically optimised for high efficiency, the best implementation is unlikely to be the most readable one. For this reason, we formalise the property of functional correctness of the `RC4` implementation in terms of input/output behavioural equivalence to another (more readable) `C` implementation of the same algorithm. We then explore techniques to prove such an equivalence, which we believe may be of independent interest.

Paper Organisation. Sections 2 and 3 give background on deductive verification and RC4 in `openSSL`. Section 4 introduces the method used to prove equivalence between the reference and practical implementations of RC4, and Sections 5 and 6 describe the formalisation of loop refactorings in Coq, based on the notion of *natural invariant*. Section 7 presents related work and Section 8 concludes the paper.

2 Background: Deduction-Based Program Verification

The techniques employed in this paper are based on Hoare Logic [12], brought to practice through the use of *contracts* – specifications consisting of preconditions and postconditions, annotated into the programs. In recent years, verification tools based on contracts have become more and more popular, as their scope evolved from toy languages to very realistic fragments of languages like C, C#, or Java.

In a nutshell, a verification infra-structure consists of a verification conditions generator (VCGen for short) and a proof tool, which may be either an automatic theorem prover or an interactive proof assistant. The VCGen reads in the annotated code (which contains contracts and other annotations meant to facilitate the verification, such as loop invariants and variants) and produces a set of proof obligations known as *verification conditions*, that will be sent to the proof tool. The correctness of the VCGen guarantees that, if all the proof obligations are valid, then the program is correct with respect to its specification. Depending on the specified properties, the verification conditions may, or may not, be automatically provable.

The concrete tools we have used in this work were `Frama-c` [3], a tool for the static analysis of `C` programs that contains a multi-prover VCGen [10]; and a set of proof tools that included the Coq proof assistant [18], and the Simplify [9] and Ergo [7] automatic theorem provers. `C` programs are annotated using the ANSI-C Specification Language (ACSL [3]). Both `Frama-c` and ACSL are work in progress; we have used the latest (Lithium) release of `Frama-c`.

`Frama-c` contains the `gwhy` graphical front-end that allows to monitor individual verification conditions. This is particularly useful when combined with the possibility of exporting the conditions to various proof tools, which allows users to first try discharging conditions with one or more automatic provers, leaving the harder conditions to be studied with the help of an interactive proof assistant. An additional feature of `Frama-c` that we have found useful is the declaration of Lemmas. Unlike axioms, which require no proof, lemmas are results that can be used to prove goals, but give themselves origin to new goals. In the proofs we developed, it was often the case that once an appropriate lemma was provided, all the verification conditions could be automatically discharged, leaving only the difficult lemma to be proved in Coq.

3 The RC4 Cipher and Its Implementation in `openSSL`

`RC4` is a symmetric cipher designed by Ron Rivest at RSA labs in 1987. It is a proprietary algorithm, and its definition was never officially released. Source code that allegedly implements the `RC4` cipher was leaked on the internet in 1994, and this is commonly known as `ARC4` due to trademark restrictions. In this work we will use the `RC4` denomination to denote the definition adopted in literature [16]. `RC4` is widely used in commercial products, as it is included as one of the recommended encryption schemes in standards such as TLS, WEP

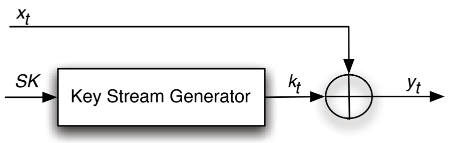

Fig. 1. Block diagram of the RC4 cipher

and WPA. In particular, an implementation of `RC4` is provided in the pervasively used open-source library `openSSL`, which we selected as the case study for this paper.

In cryptographic terms, `RC4` is a synchronous stream cipher, which means that it is structured as two independent blocks, as shown in Figure 1. The security of the `RC4` cipher resides in the strength of the key stream generator, which is initialized with a secret key SK. The key stream output is a byte[1] sequence k_t that approximates a perfectly random bit string, and is independent of plaintext and ciphertext. The encryption operation consists simply of XOR-ing each plaintext byte x_t with a fresh keystream byte k_t. Decryption operates in an identical way. The key stream generator operates over a state which includes a permutation table $S = (S[l])_{l=0}^{l=255}$ of (unsigned) byte-sized values, and two (unsigned) byte-sized indices i and j. We denote the values of these variables at time t by S_t, i_t and j_t. The state and output of the key stream generator at time t (for $t \geq 1$) are calculated according to the following recurrence, in which all additions are carried out modulo 256.

$$\begin{aligned} i_t &= i_{t-1} + 1 \\ j_t &= j_{t-1} + S_{t-1}[i_t] \\ S_t[i_t] &= S_{t-1}[j_t] \\ S_t[j_t] &= S_{t-1}[i_t] \\ k_t &= S_t[S_t[i_t] + S_t[j_t]] \end{aligned}$$

The initial values of the indices i_0 and j_0 are set to 0, and the initial value of the permutation table S_0 is derived from the secret key SK. The details of this initialisation are imaterial for the purpose of this paper, as they are excluded from the analysis.

We present in Appendix A the `C` implementation of RC4 included in the `openSSL` open-source. The function receives the current state of the RC4 key stream generator (`key`), and two arrays whose length is provided in parameter `len`. The first array contains the plaintext (`indata`), and the second array will be used to return the ciphertext (`outdata`). We note that this implementation is much less readable than the concise description provided above, as it has been optimised for speed using various tricks, including macro inlining and loop unrolling.

[1] We adopt the most widely used version of RC4 which operates over byte-sized words, which is also the one implemented in `openSSL`.

```
unsigned char RC4NextKeySymbol(RC4_KEY *key) {
  unsigned char tx,ty;

  key->x=(key->x+1) % 256;
  tx=key->data[key->x];
  key->y=(tx+key->y) % 256;
  ty=key->data[key->y];
  key->data[key->x]=ty;
  key->data[key->y]=tx;
  return key->data[(tx+ty) % 256];
}

void RC4(RC4_KEY *key, const unsigned long len,
         const unsigned char *indata, unsigned char *outdata) {
  int i=0;
  while(i<len) { outdata[i]=indata[i] ^ RC4NextKeySymbol(key); i++; }
}
```

Fig. 2. RC4 specification

4 Functional Correctness of Code Refactoring

It is typical of cryptographic software that specifications are given as algorithms, rather than using the notion of an abstract model. The programmer is free to improve the code, say by introducing optimizations or internal reorganizations (e.g. to improve efficiency, maintainability or to satisfy non-functional security properties), as long as the input-output behaviour is the same as that prescribed by a reference implementation. In software engineering, such a transformation is usually known as *code refactoring*.

To illustrate this point, recall the description of the RC4 algorithm provided in Section 3. A direct transcription of this specification to a C implementation could look something like the code in Figure 2. Although this implementation is quite readable, and arguably verifiable by inspection, it was created without the slightest consideration for efficiency. This stands in contrast with the openSSL implementation of RC4 (see Appendix A) where readability (and the inherent assurance of correctness) was sacrificed to achieve better performance.

This example supports the domain-specific motivation for the discussion presented in this section: the natural way to obtain assurance that an implementation of a cryptographic algorithm is correct, is to verify that it is functionally equivalent to another (more readable) implementation of the same algorithm. We have investigated how this goal can be achieved for the particular case of RC4, by identifying refactoring steps that may require a proof of equivalence in order to establish the correctness of different RC4 implementations. We describe these refactoring steps in the remainder of this section. In the next section we present our approach to verifying the identified class of equivalence relations using an off-the-shelf tool such as Frama-c. The results we obtain are, of course,

```
void RC4(RC4_KEY *key, const unsigned long len,
              const unsigned char *indata, unsigned char *outdata) {
  unsigned char keystream[len];

  int i=0;
  while(i<len) { keystream[i] = RC4NextKeySymbol(key); i++; }

  i=0;
  while(i<len) { outdata[i]=indata[i] ^ keystream[i]; i++; }
}
```

Fig. 3. RC4 implementation with key pre-processing

not only applicable to implementations of other cryptographic algorithms, but also to other application domains where similar program transformations may be employed.

A simple refactoring to capture key pre-processing. The first example we present of a possible refactoring of the RC4 specification in Figure 2 is suggested by a common optimisation performed when using stream ciphers. Indeed, one of the ways of speeding up the throughput of stream cipher processing is to compute (a portion of) the key stream before the plaintext is available (or the ciphertext if one is decrypting). This means that the encryption operation to be performed on-the-fly is then reduced to simple masking using an `XOR` operation, which can be done extremely fast. For sychronous ciphers such as RC4, the number of key stream bits that can be pre-computed can be arbitrarily large, as this is totally independent of the encrypted data. The version of RC4 presented in Figure 3 moves in this direction by separating the key stream generation process from the plaintext masking (or ciphertext unmasking process). In the next section we will discuss a technique that can be used to prove equivalence beween the programs in Figures 2 and 3 using a verification infrastructure like that discussed in Section 2.

A sequence of refactorings leading to the `openssl` implementation. We now discuss a more elaborate sequence of refactoring steps that permit reaching the `openSSL` implementation of RC4 in Appendix A, departing from the reference implementation in Figure 2. The first refactoring step, leading to the RC4 function in Figure 4, top, is not very interesting from a verification point of view. It consists of a number of simple transformations whose validity can be proven with some effort using `Frama-c`: (1) removing the auxiliary function by inlining the corresponding code in the main function body; (2) rearranging local variables to match those in the `openSSL` implementation; (3) applying the transitivity property of assignments in C to combine two statements; and (4) replacing modular operations by their equivalent bit-wise operations. A macro is also introduced to improve readability.

```
void RC4(RC4_KEY *key, const unsigned long len,
     const unsigned char *indata, unsigned char *outdata)
{
  unsigned char x,y,tx,ty, *d;
  int i;

  x = key->x; y = key->y; d = key-> data;

  i=0;
  while(i<len) { RC4LOOP(indata,outdata,i); i++; }

  key->x=x; key->y=y;
}
```

```
void RC4(RC4_KEY *key, const unsigned long len,
         const unsigned char *indata, unsigned char *outdata)
{
  unsigned char x,y,tx,ty, *d;
  int i;

  x = key->x; y = key->y; d = key-> data;

  i= (int)(len>>3L);
  while(i>0) {
    RC4LOOP(indata,outdata,0);
    RC4LOOP(indata,outdata,1);
    RC4LOOP(indata,outdata,2);
    RC4LOOP(indata,outdata,3);
    RC4LOOP(indata,outdata,4);
    RC4LOOP(indata,outdata,5);
    RC4LOOP(indata,outdata,6);
    RC4LOOP(indata,outdata,7);
    indata+=8; outdata+=8; i--;
  }

  i=(int)(len&0x07);
  while(i>0) {RC4LOOP(indata,outdata,i); i--; }

  key->x=x; key->y=y;
}
```

Fig. 4. RC4 refactoring steps 1 (top) and 2 (bottom)

The next refactoring steps, leading to the version shown in Figure 4, bottom, are more interesting examples of transformations involving loop refactorings. Concretely, the main loop is first separated into two loops with the same body, which are sequentially composed to realise the original number of iterations. The first loop is then modified by explicitly composing the original body with itself 8 times, and altering the increments accordingly.

The final refactoring steps, leading to the `openssl` version of RC4 in Appendix A, are introduced to achieve additional speed-ups. Firstly, pointer arithmetic is used to reduce the range of indexing operations, and loop counting is inverted. Then, different control flow constructions are applied: all `while` loops are reformulated using the `break` statement to remove the final backward jump, and `if` constructions are introduced to detect termination cases. Again, these refactoring steps can be handled in `Frama-c` with some effort, but they do not require non-trivial proof steps that justify a detailed presentation.

In the remainder of this paper we concentrate on presenting a technique that can be used to prove the equivalence of the different versions of the RC4 function that spring from the specific loop transformations outlined in the second step above.

Equivalence by Composition. We now formalise a notion of program equivalence that permits dealing with the refactoring paradigm introduced above. The required notion of program equivalence is based on Hoare logic, using a program composition technique inspired by self-composition, a technique for reasoning axiomatically about non-interference properties of programs [2]. The general technique introduced here sets the grounds for the work presented in the next section, where we explore the technical details involved in proving program equivalence relations such as those arising from the refactorings described above for RC4.

The basic principle underlying self-composition can be adapted to the current context: given two terminating programs C_1 and C_2, they can be combined by first renaming the variables in one of the programs so that they use distinct name spaces, and then composing the programs sequentially. Given some program C, let C^s be the program that is equal to C except that every variable x is renamed to a fresh variable x^s.

Let V be the set of variables occurring in both programs. The idea that we want to capture is that if the programs are executed from indistinguishable states with respect to V, they terminate in states that are also indistinguishable. C_1 and C_2 will be defined as equivalent if every execution of the composed program $C_1; C_2^s$, starting from a state in which the values of corresponding variables are equal, terminates in a state with the same property. This can be expressed as the following Hoare logic total correctness specification, that can be expressed in ACSL.

$$\left[\bigwedge_{x\in V} x = x^s\right] \; C_1; C_2^s \; \left[\bigwedge_{x\in V} x = x^s\right]$$

Weaker notions of equivalence can be handled by taking V to be a subset of $\mathit{Vars}(C_1) \cap \mathit{Vars}(C_2)$.

5 Proving Equivalence Using Natural Invariants

Is this section we elaborate on the general approach that we adopt to prove the equivalence between a refactored version of a function such as those in Figure 4, with respect to the originating function, in this case the reference implementation in Figure 2. In order to establish this equivalence using a deductive framework such as `Frama-c`, we need to:

– create a composed program which aggregates the two versions of the original program we aim to prove functionally equivalent;
– annotate the composed program with appropriate contracts and loop invariants;
– discharge the resulting proof obligations.

Moreover, we would like to overcome these steps with a reasonable degree of automation. Here, *reasonable* essentially means that we intend to take the maximum advantage from the fact that we are dealing with program refactorings, which admittedly share most of its control structure. Our strategy for tackling these problems consist in: (1) extracting a relational specification directly from the program code; (2) annotating the program with invariants derived from the specification; (3) generating specific lemmas justifying the most significant refactorings; and (4) using an automatic first-order theorem prover to discharge the proof-obligations. The generated lemmas, which constitute the (small) non-trivial part of the proof, must then be justified using an interactive theorem prover.

To illustrate this methodology, we consider a simple *While-language* with integer expressions and arrays. Its syntax is given by:

$$\begin{aligned}
P &::= \{P\} \mid \textbf{skip} \mid P_1; P_2 \mid V := E_{int} \mid A[E_{int}] := E_{int} \\
&\quad \mid \textbf{if } (E_{bool}) \textbf{ then } P_1 \textbf{ else } P_2 \mid \textbf{while } (E_{bool})\ P \\
E_{int} &::= Const_{int} | E_{int}\ op\ E_{int} | A[E_{int}] | ... \\
E_{bool} &::= true | false | E_{bool} \wedge E_{bool} | E_{bool} \vee E_{bool} | E_{int}\ opRel\ E_{int}
\end{aligned}$$

For simplicity we do not include any form of variable declaration. Instead, we consider a fixed `State` type to keep track of all the variable values during the execution of the program. Integer variables are interpreted as (unbound) integers and arrays as functions from integers to integers (no size/range checking). We also adopt the usual axioms for array access and update operations.

$$\begin{aligned}
\text{access} &: (Z \to Z) \times Z \to Z \\
\text{update} &: (Z \to Z) \times Z \times Z \to (Z \to Z) \\
&\text{access}(\text{update}(a, k, x), k) = x \\
&\text{access}(\text{update}(a, k', x), k) = \text{access}(a, k) \qquad \text{, if } k \neq k'.
\end{aligned}$$

The `State` type is defined as the cartesian product of the corresponding interpretation domains (each variable is associated with a particular position). We also consider an equivalence relation $\equiv$ that captures two extensionally equal states. Integer and boolean expressions are interpreted in a particular state, that is $[\![e_{Int}]\!] : State \rightarrow Z$, $[\![e_{Bool}]\!] : State \rightarrow B$. We take the standard definition for the big-step semantics of a program as its *natural specification.* Concretely:

$$\begin{aligned}
\mathsf{spec}_{\mathbf{skip}}(s,s') &= s \equiv s' \\
\mathsf{spec}_{\{P\}}(s,s') &= \mathsf{spec}_P(s,s') \\
\mathsf{spec}_{P_1;P_2}(s,s') &= \exists s'',\ \mathsf{spec}_{P_1}(s,s'') \wedge \mathsf{spec}_{P_2}(s'',s') \\
\mathsf{spec}_{v:=E}(s,s') &= s' \equiv s\{v \leftarrow [\![E]\!](s)\} \\
\mathsf{spec}_{a[E_1]=E_2}(s,s') &= s' \equiv s\{a \leftarrow \mathrm{update}(a,[\![E_1]\!](s),[\![E_2]\!](s))\} \\
\mathsf{spec}_{\mathbf{if}\ (C)\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2}(s,s') &= ([\![C]\!]s \wedge \mathsf{spec}_{P_1}(s,s')) \vee (\neg[\![C]\!](s) \wedge \mathsf{spec}_{P_2}(s,s')) \\
\mathsf{spec}_{\mathbf{while}\ (C)\ P}(s,s') &= \exists n,\quad \mathsf{loop}^n_{C,\mathsf{spec}_P(s,s')}(s,s') \wedge \neg[\![C]\!](s')
\end{aligned}$$

where $\mathsf{loop}^n_{C,R}(s,s')$ is the inductively defined relation

$$\begin{aligned}
\mathsf{loop}^0_{C,R}(s,s') &\Longleftarrow s \equiv s' \\
\mathsf{loop}^{S(n)}_{C,R}(s,s') &\Longleftarrow \exists s'',\ \mathsf{loop}^n_{C,R}(s,s'') \wedge [\![C]\!](s'') \wedge R(s'',s')
\end{aligned}$$

The relation $\mathsf{loop}^n_{C,R}(s,s')$ denotes the loop specification for the body R under condition C. In this definition we have made explicit the *iteration rank* (iteration count) in superscript – in fact, we will see that it is often convenient to consider it explicitly in the proofs. Neverthless, when omitted, it should be considered as existentially quantified. Also, we will omit subscripts (both in loop and spec) when the corresponding programs are clear from the context. From the loop relation we recover what we call the loop's *natural invariant* as:

$$\mathrm{Inv}_{loop}(s) = \mathsf{loop}_{C,R}(s@\mathrm{Init}, s)$$

where C and R are the loop's condition and body, respectively, and $s@\mathrm{Init}$ denotes the snapshot of the loop's initial state.

Expressiveness and Relative Completeness. Natural invariants depend on a sufficiently expressive assertion language that should allow defining new inductive relations. This corresponds essentially to Cook's expressiveness criteria in his relative completeness result for Hoare Logic [8]. In fact, from the definition of spec we can easily recover the *strongest liberal predicate* as

$$\mathrm{slp}(S,P) = \{s' \mid P(s) \wedge \mathsf{spec}_S(s,s')\}$$

An immediate consequence of this observation is that we might conduct the verification of an arbitrary Hoare triple logically, namely

$$\{P\}S\{Q\} \qquad \text{iff} \qquad \mathrm{slp}(S,P) \supseteq Q \qquad \text{iff} \qquad P(s) \wedge \mathsf{spec}_S(s,s') \wedge Q(s').$$

Note that these requirements surpass the realm of first-order logic. Thus, when the target verification tool is a first-order theorem prover, we shall rely on a weak axiomatisation of these predicates (in our case, we consider an axiom for each constructor and a simple inversion principle). This observation clarifies why we need to supplement the first-order theory with additional lemmas. Moreover, it also shows that failure in verification is not necessarily caused by limitations in the first-order theorem provers. When full-fledged inductive reasoning is needed, we resort to Coq's higher-order logic capabilities to interactively prove specific lemmas. Fortunately, it is possible to identify a set of general lemmas that can be proven once-and-for-all, and that permit justifying interesting refactorings.

General Properties. We call *plain theory* to the first-order theory resulting from the program specification and the corresponding weak axiomatization of loop predicates. A consequence of the limitations of first-order theory/provers mentioned above is that the plain theory is insufficient to establish the adequacy of even the most trivial refactoring (the identity refactoring) when the programs use loops. To illustrate what is missing, we need to walk the reader through the proof of the following spec properties:

Proposition 1. *For every program fragment P and states s_1, s_2, s'_1, s'_2,*

- spec *preserves* $\equiv$, *i.e.* $s_1 \equiv s_2 \wedge s'_1 \equiv s'_2 \wedge \mathsf{spec}_P(s_1, s'_1) \Rightarrow \mathsf{spec}_P(s_2, s'_2)$.
- spec *is deterministic, i.e.* $\mathsf{spec}_P(s, s'_1) \wedge \mathsf{spec}_P(s, s'_2) \Rightarrow s'_1 \equiv s'_2$.

The proof follows straight by induction on the program P using the following lemma

Lemma 1. *Let $R(s, s')$ be a deterministic relation on states, and C a boolean condition. Then,* $\mathsf{loop}_{C,R}(s, s')$ *is deterministic whenever* $\neg[\![C]\!](s')$.

This in turn is a consequence of the following two assertions:

$$s_1 \equiv s_2 \wedge \mathsf{loop}^{n_1}_{C,R}(s_1, s'_1) \wedge \neg[\![C]\!](s'_1) \wedge \mathsf{loop}^{n_2}_{C,R}(s_2, s'_2) \wedge \neg[\![C]\!](s'_2) \Longrightarrow n_1 = n_2$$
$$s_1 \equiv s_2 \wedge \mathsf{loop}^{n}_{C,R}(s_1, s'_1) \wedge \mathsf{loop}^{n}_{C,R}(s_2, s'_2) \Longrightarrow s'_1 \equiv s'_2$$

Both of these statements are directly proved by a simple induction (on $\max(n_1, n_2)$ in the first case, and on n in the second). The first statement establishes the *synchronization* of both executions and the second their *determinism*. Augmenting the plain theory with these lemmas is mandatory to perform even the most basic reasoning.

This factorization strategy, in which we detach the synchronization and determinism properties, underlies our proposed paradigm for reasoning about multiple executions of the same (or related) program fragments. Moreover, we can strengthen the synchronization lemma by observing that it only depends on the equivalence between fragments of the initial state, namely those that affect the loop's condition. The determinism lemma can itself be rephrased replacing state equivalence by an arbitrary predicate.

Justifying Loop Refactorings. For the sake of presentation, we restrict our attention to specifications obtained from single loops with loop-free bodies. That is, we consider natural invariants of the form:

$$\mathsf{loop}_{C,\mathsf{spec}(P)}(s, s')$$

where P contain no loops. This scenario is enough to illustrate the applicability of the proposed strategy in tackling the sort of program refactorings needed for establishing correctness of the RC4 `openSSL` implementation.

The simplest loop refactoring that we can address using our technique is *loop unrolling*, where we detach instances of the loop-body. We find this refactoring in the optimisations described in the previous sections. This sort of transformation is justified by the loop's inversion lemma:

$$\forall n\ s\ s',\ \mathsf{loop}^{S(n)}_{C,R}(s,s') \Longrightarrow \exists s'', \quad \mathsf{loop}^{n}_{C,R}(s,s'') \wedge [\![C]\!](s'') \wedge R(s'',s').$$

This relatively simple class of refactorings can then be handled directly by the plain theory augmented with synchronization and determinism lemmas.

For more interesting refactorings, we may need to formulate specific lemmas to justify them. Let us illustrate this by a *loop-fusion* refactoring: we consider the equivalence between two consecutive loops (loops 1 and 2) and one single *fused* loop (loop 3). This is applicable to the RC4 pre-processing optimisation presented in the previous section. Let us denote the inductive predicates of these loops by $\mathsf{loop}_1, \mathsf{loop}_2$ and loop_3, respectively. We assume, for simplicity, that all the loops share the same control structure (loop condition and associated state). This means that we are able to prove *mixed* synchronization lemmas such as, for all $n_1\ n_2\ s_1\ s_2\ s'_1\ s'_2$,

$$\pi^C(s_1) \equiv \pi^C(s_2) \wedge \mathsf{loop}^{n_1}_1(s_1,s'_1) \wedge \neg[\![C]\!](s'_1) \wedge \mathsf{loop}^{n_2}_2(s_2,s'_2) \wedge \neg[\![C]\!](s'_2) \Longrightarrow n_1 = n_2.$$

Again, the proof is a straightforward generalisation of the single loop version. Once this result is estabished, one can move to the proof of the main lemma that can be used to justify the fusion refactoring:

$$\begin{aligned}&\forall n\ s_1\ s_2\ s'_1\ s''_1\ s'_2,\\ &\qquad s_1 \equiv s_2 \wedge \mathsf{loop}^n_1(s_1,s''_1) \wedge \mathsf{loop}^n_2(s''_1,s'_1) \wedge \mathsf{loop}^n_3(s_2,s'_2) \Longrightarrow s'_1 \equiv s'_2.\end{aligned}$$

The advantage of our method is that, since this lemma is based on simple properties concerning the three loop bodies, which are all non-recursive, it can be easily discharged by automatic provers.

6 Implementation Details

We have tested the proposed methodology in checking the correctness of the RC4 `openSSL` implementation (shown in appendix). The specification predicates were extracted manually, and included in the ACSL code as inductive definitions. These definitions are allowed by the last revision of the ACSL language (version 1.4), but we remark that when the target verification tool is a first-order prover they are translated to a weak axiomatization (as described in Section 5). Additional lemmas were also included in the ACSL code and proved by the Coq proof assistant. For that purpose, we have developed a library that includes a full formalization of natural invariants as presented in the last section. This library makes extensive use of the Coq's module system [6] in order to structure the development. As a rule, we embed each lemma and respective proof in a functor parametrized by the basic facts it depends on. In particular, we have defined functors for deriving synchronization, determinism and loop fusion lemmas. All the facts required by these functors are non-iterative, and thus are easily discharged by the automatic provers. In this way, we are able to treat this library as a catalog of refactorings that can be used on demand during the verification process — we emphasise that there is no need to conduct further interactive proofs, unless this catalog is extended to cover a new class of loop refactorings.

7 Related Work

Natural Invariants provide an explicit rendition of program semantics. In [13] a similar encoding of program semantics in logical form can be found, which advocates the use of second-order logic as appropriate to reason about programs, since it allows to capture the inductive nature of the input-output relations for iterative programs. To some extent, our use of Coq's higher-order logic may be seen as an endorsement of that view. However, we have made an effort to combine the strength of higher-order logic reasoning with facilities made available by automatic first-order provers.

Our "proof-by-composition" technique is reminiscent of the self-composition approach for verifying non-interference[2]. Terauchi and Aiken [17] identified problems in applying it, arguing that automatic tools (software model checkers like SLAM [1] and BLAST [11]) are not powerful enough to verify this property over programs of realistic size. To compensate for this, the authors propose a program transformation technique, which incorporates the notion of security level downgrading using *relaxed non-interference* [14]. Our work proposes an alternative solution since it enriches the uderlying first-order theory with lemmas that overcome the identified limitations.

Relational Hoare Logic [4] has also been used to prove the soundness of program analyses and optimising transformations. Its scope is thus similar to our proofs-by-composition setting. The main difference is the fact that we do not need to move away from traditional Hoare Logic, which allows us to rely on standard available verification tools.

8 Conclusions

In this paper we have presented a methodology for verifying correctness of implementations with regard to reference implementations, an important concern in domains such as the verification and certification of cryptographic software implementations. We have focused on proposing strategies and techniques allowing us to maximize the benefits of using well established and publicly available tools, such as `Frama-c`, first-order automatic theorem provers and the Coq proof assistant. The approach can be summed up as follows

1. Program equivalences in general can be expressed (for terminating programs) as Hoare triples using a composition technique that simulates the execution of two programs by a single program. Such triples can be written in an interface specification language like ACSL and fed to a standard VCGen like `Frama-c`.
2. However, program equivalences are difficult verification challenges by nature, and automatic proof is, on its own, of little help. Resorting to an interactive proof tool to conduct inductive proofs involving loops is inevitable.
3. Natural invariants are good candidates for establishing the connection between the interface specification language and the proof assistant: on one hand, all the interactive reasoning is transferred to the inductive predicates that form the invariant; on the other hand, the invariant can be annotated into the specification files to be fed through the VCGen. We remark that these invariants (and some standard lemmas) can be generated mechanically.
4. Concluding the verification process is then a matter of identifying the relevant refactoring and instantiating the corresponding lemma. Once equipped with these lemmas an automatic prover is able to discharge the remaining proof obligations.
5. Once recognized, a new refactoring might be included by defining a new functor responsible for instantiating the corresponding lemma. It will require a once-and-for-all formal proof asserting the refactoring correctness (proved interactively in Coq).

This approach was put to practice to prove (as a sequence of refactoring steps) the equivalence between a reference implementation of an open-source cryptographic algorithm and the realistic implementation included in the appendix. Other applications that we are developing for this approach based on natural invariants include proofs of information flow security properties, using the self-composition technique, and related properties such as the absence of error propagation in stream ciphers.

References

1. Ball, T., Rajamani, S.K.: The slam project: debugging system software via static analysis. In: POPL 2002: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 1–3. ACM, New York (2002)
2. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSFW, pp. 100–114. IEEE Computer Society, Los Alamitos (2004)
3. Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specfication Language. In: CEA LIST and INRIA, 2008. Preliminary design (version 1.4), December 12 (2008)
4. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) POPL, pp. 14–25. ACM, New York (2004)
5. Computer Aided Cryptography Engineering. EU FP7, `http://www.cace-project.eu/`

6. Chrzaszcz, J.: Implementation of modules in the Coq system. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 270–286. Springer, Heidelberg (2003)
7. Conchon, S., Contejean, E., Kanig, J.: Ergo: a theorem prover for polymorphic first-order logic modulo theories (2006)
8. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. 7(1), 70–90 (1978)
9. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
10. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 58–70. ACM, New York (2002)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12, 576–580 (1969)
13. Leivant, D.: Logical and mathematical reasoning about imperative programs. In: POPL, pp. 132–140 (1985)
14. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 158–170. ACM Press, New York (2005)
15. The OpenSSL Project, `http://www.openssl.org`
16. Schneier, B.: Applied cryptography: protocols, algorithms, and source code in C, 2nd edn. Wiley, New York (1996)
17. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
18. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.2 (2008), `http://coq.inria.fr`

A openSSL Implementation of RC4

```
typedef struct rc4_key_st { unsigned char x,y,data[256];} RC4_KEY;

void RC4(RC4_KEY *key,const unsigned long len,
         unsigned char *indata, unsigned char *outdata) {

  register unsigned char *d,x,y,tx,ty;
  int i;
  x=key->x;
  y=key->y;
  d=key->data;

#define LOOP(in,out) \
  x=((x+1)&0xff); \
  tx=d[x]; \
  y=((tx+y)&0xff); \
  d[x]=ty=d[y]; \
  d[y]=tx; \
  (out) = d[((tx+ty)&0xff)]^ (in);
#define RC4_LOOP(a,b,i) LOOP(a[i],b[i])

  i=(int)(len>>3L);
  if (i) {
    while(1) {
      RC4_LOOP(indata,outdata,0);
      RC4_LOOP(indata,outdata,1);
      RC4_LOOP(indata,outdata,2);
      RC4_LOOP(indata,outdata,3);
      RC4_LOOP(indata,outdata,4);
      RC4_LOOP(indata,outdata,5);
      RC4_LOOP(indata,outdata,6);
      RC4_LOOP(indata,outdata,7);
      indata+=8;
      outdata+=8;
      if (--i == 0) break;}}
  i=(int)(len&0x07);
  if(i) {
    while(1) {
      RC4_LOOP(indata,outdata,0); if (--i == 0) break;
      RC4_LOOP(indata,outdata,1); if (--i == 0) break;
      RC4_LOOP(indata,outdata,2); if (--i == 0) break;
      RC4_LOOP(indata,outdata,3); if (--i == 0) break;
      RC4_LOOP(indata,outdata,4); if (--i == 0) break;
      RC4_LOOP(indata,outdata,5); if (--i == 0) break;
      RC4_LOOP(indata,outdata,6); if (--i == 0) break;}}
  key->x=x;
  key->y=y;
}
```