View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Deductive Verification of Cryptographic Software

José Bacelar Almeida · Manuel Barbosa · Jorge Sousa Pinto · Bárbara Vieira
E-mail: (jba,mbb,jsp,barbarasv)@di.uminho.pt

Abstract We apply state-of-the art deductive verification tools to check security-relevant properties of cryptographic software, including safety, absence of error propagation, and correctness with respect to reference implementations. We also develop techniques to help us in our task, focusing on methods oriented towards increased levels of automation, in scenarios where there are clear obvious limits to such automation. These techniques allow us to integrate automatic proof tools with an interactive proof assistant, where the latter is used off-line to prove once-and-for-all fundamental lemmas about properties of programs. The techniques developed have independent interest for practical deductive verification in general.

Keywords Cryptographic algorithms; program verification; program equivalence; self-composition.

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.

– Firstly, cryptography is an inherently interdisciplinary 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 be optimised aggressively, without altering the security semantics.
– Finally, typical software engineers develop systems focused on desktop class processors within computers in our offices and homes. The special case of cryptographic software is implemented on a 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 cryptographic software engineers understand each platform and the related security requirements, they must also optimise each algorithm with respect to each platform, since each one will have vastly different performance characteristics.

Program Verification is the area of Formal Methods that attempts to check properties of software statically, with the help of an axiomatic semantics of the underlying programming language and a proof tool. Specifically, we are interested in techniques based on Hoare logic [20], 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[18,6], C# [4], or Java[23]. We will use the expression *deductive program verification* to distinguish this approach from other ways of checking properties of pro-

J.B. Almeida · M. Barbosa · J.S. Pinto · B. Vieira
CCTC / Departamento de Informática
Universidade do Minho
Campus de Gualtar, 4710-Braga, Portugal

grams, such as *software model checking* [2, 19, 22]. The goal of this paper is to apply deductive program verification techniques to prove diverse properties of cryptographic software.

Contributions. We describe results obtained in our exploration of existing verification techniques and tools (used to construct high-assurance software implementations in other domains) to the concrete case of cryptographic software. In doing so we have also developed techniques that are of independent interest. Our contributions are the following.

- We propose a *composition*-based methodology for proving the functional equivalence of programs. This is inspired by the *self-composition* technique [5] that can be used to prove information flow properties of programs. Our methodology also enables self-composition proofs, although it targets the more general problem of proving the correctness of concrete implementations with respect to specifications given as reference implementations
- We employ *natural invariants* as a device to establish a correspondence between (annotation-level) axiomatic properties of programs and (a formalisation of) their operational semantics. This device take us beyond the usual scope of contract-based verification, enabling us to obtain automatic proofs relying on a battery of lemmas which are interactively proved once-and-for-all.
- We show how natural invariants are useful for reasoning about pairs of programs with similar control structures. In particular, this is a useful technique to enable program equivalence proofs in practice. It also allows for the automation of the self-composition technique, which has been identified as a major problem [32].
- We show how these results enable us to use an off-the-shelf verification tool to reason about functional correctness, safety properties, and security properties of a `C` implementation of the RC4 encryption scheme, included in the well-known open-source library `openSSL` [29].

CACE (Computer Aided Cryptography Engineering [8]) 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 toolbox will address particular software development problems and processes; and combined use of the constituent tools is enabled by designed integration between their interfaces. It is a three-year project that started in 2008.

This article stems from CACE - Work Package 5, which aims at adding formal methods technology to the tool-box, as a means to increase the degree of assurance than can be provided by the development process.

Organisation. Section 2 introduces the area of cryptographic software implementation, and identifies important security properties that deserve attention from a formal verification point of view. We then discuss in Section 3 methods to formalise and verify the validity of these properties using a deductive verification platform. Section 4 describes the development of an infrastructure to support the automation of the proposed approach, and Section 5 shows its application to a concrete case study: the verification of the RC4 `openSSL` implementation. We conclude the paper with a discussion of related work in Section 6 and some concluding remarks in Section 7.

2 A Catalogue of Software Properties

2.1 Functional Correctness

The goal of functional correctness verification is to establish that a program performs according to some intended specification. More precisely, that the input/output behaviour of the implementation matches that of the specification. This is certainly the primary concern in program verification and the context in which the deductive approach has been most widely used. Verifying functional correctness within a deductive framework typically involves the following steps:

1. annotating the source code with specification contracts;
2. adding invariant information for loops (and possibly also variants if total correctness is a concern);
3. producing, with the help of a verification condition generator tool (VCGen), a set of verification conditions; and
4. discharging them (i.e. proving them) using an automatic or interactive prover.

The critical points are steps 2 and 4, since the user needs to identify how certain properties are approximated during the loop execution – this is where most of the user activity should be focused, since richer invariants will be harder to prove, but will simplify the verification of the contract (by providing a richer set of hypotheses). Indeed, an active research area is *invariant synthesis*, which attempts to automatically generate these invariants by analysing the program. However, it should be kept in mind that the problem is inherently difficult, even if certain approaches seem to behave quite well for specific domains.

Correctness with respect to a Reference Implementation. The standard scenario in deductive verification is that specifications are written as contracts on the function and procedure interfaces. This may involve properties of the output values (typically written in first-order logic), as well as relations between input and output values.

In this work we take an alternative route. We are interested in verifying cryptographic software, whose specifications are typically given as operational descriptions (i.e. as

algorithms). This is the case, for example, in symmetric-key techniques such as ciphers, message authentication codes, cryptographic hash functions, etc. When implementing such a technique, the programmer will follow this description, but is free to improve the code, say by introducing optimisations or internal reorganisations (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 the specification.

To some extent, the specification acts as a reference implementation: verifying functional correctness is reduced to proving program equivalence. Again, this is a difficult (and undecidable) problem, although in this concrete application domain we can rely on the fact that implementation and specification share most of their internal structure (since the latter has been adopted as a model for the former). Indeed, the sort of equivalence proof required for cryptographic software corresponds to what is usually known in software engineering as *code refactoring*.

2.2 Safety Properties

Due to the inherent difficulty of the verification of functional properties, less ambitious forms of verification are often used. A widespread verification approach which aims at increasing the level of assurance that can be placed on software implementations is to confine the analysis to a restricted class of properties that rule out the occurrence of some recognisable "bad things". This is what is called a *safety analysis*, which often includes properties associated with common vulnerabilities arising from coding errors such as de-referencing invalid pointers, accessing containers with invalid indexes, calculation errors due to overflows, etc.

The advantage of focusing on such simple properties is that a significant degree of automation can be achieved, minimising user intervention and impact on development time. The resulting level of assurance, far from being absolute, is nevertheless sufficient for a wide class of application scenarios. In this work, we also briefly review how a deductive verification tool can be used to perform this sort of analysis.

2.3 Information Flow Security

Information flow security refers to a class of security policies that constrain the ways in which information can be manipulated during program execution. These properties can be formulated in terms of *non-interference* between high-confidentiality input variables and low-confidentiality output variables, and are usually verified using a special extended type system [36,26,3]. A dual formulation permits capturing security policies that constrain information flow from non-trustworthy (or low-integrity) inputs, to trusted (or high-integrity) outputs. In Section 6 we provide an overview of developments in this area related to the work in this paper.

Consider the more common case of secure information-flow that aims at preserving data confidentiality. Information may flow from high-security to low-security variables either directly via assignment instructions, or indirectly. The following code from Terauchi and Aiken [32] computes in `f1` the `n`$^{\mathtt{th}}$ Fibonacci number and then assigns a value to `l` that depends on the value of `f1`.

```
f1 = 1; f2 = 0;
while (n > 0) {
  f1 = f1 + f2;
  f2 = f1 - f2;
  n--; }
if (f1 > k) l = 1; else l = 0;
```

Let `l` be low security and `n` high security; then clearly there is an indirect information leakage from `n` to `l`, since the assignment `l = 1` is guarded by a condition that depends on the value of `f1`, and assignments to the latter variable are performed inside a loop that is controlled by the high security condition $\mathtt{n} > 0$. The program is thus insecure. If `n` were not high security, the program would of course be secure.

Type-based analyses would address the problem by tracking assignments to low security variables. Observe, however, that this fails to capture subtle situations where an apparently insecure program is in fact secure. If the last line of the program were changed to

```
if (f1 > k) l = E1; else l = E2;
```

where `E1`, `E2` are two expressions that evaluate to the same value, then the program should be classified as high security, since there is no way to tell from the final value of `l` anything about `f1`. Type-based analyses would typically fail to distinguish this from the previous program: both would be conservatively classified as insecure. An alternative approach is to define a program as secure if different terminating executions, starting from states that differ only in the values of high-security variables, result in final states that are equivalent with respect to the values of low-security variables. This approach, based on the language semantics, avoids the excessively conservative behaviour of the previous method.

More formally, let V_H and V_L denote respectively the sets of high-security and low-security variables of C, and $V_L' = \mathit{Vars}(C) \setminus V_H$. We write $(C, \sigma) \Downarrow \tau$ to denote the fact that when executed in state σ, C stops in state τ (states are functions mapping variables to values; $\Downarrow$ is the evaluation relation in a big-step semantics of the underlying language). Then the program C is secure if for arbitrary states σ, τ,

$$\sigma \stackrel{V_L'}{=} \tau \,\wedge\, (C,\sigma) \Downarrow \sigma' \,\wedge\, (C,\tau) \Downarrow \tau' \implies \sigma' \stackrel{V_L}{=} \tau'$$

where $\sigma \stackrel{X}{=} \tau$ denotes the fact that $\sigma(x) = \tau(x)$ for all $x \in X$, i.e. σ and τ are X-indistinguishable.

Variants and Other Uses of Non-Interference. Non-interference has been recognised to be very strict in the sense that it excludes any form of information flow between high level and low level security variables. Most of the times one needs some mechanism for *declassifying* data, allowing controlled flux of information between security levels. In Section 6 we point to some related work in this direction.

We also point out that non-interference may be useful to express properties that are not directly concerned with security, but are nevertheless useful in characterising specific aspects of cryptographic algorithms. As an example, consider the *error-propagation* property of stream ciphers such as RC4, describing how they behave when used to transfer data over channels which may introduce transmission errors.

The way in which the decryption process reflects a wrong ciphertext symbol in the resulting plaintext is relevant: depending on the encryption scheme construction, a ciphertext error may simply lead to a corresponding flip in a plaintext symbol, or it may affect a significant number of subsequent symbols. This property, sometimes called error propagation, is usually taken as a criterion for selecting ciphers for noisy communication media, where the absence of error propagation can greatly increase throughput. Note that error propagation can sometimes be seen as a desirable feature, as it amplifies errors that may be introduced maliciously, making them easier to detect.

The intuition underlying the formalisation of error-propagation with non-interference is that secure information flow can be guaranteed by checking that arbitrary changes in low-integrity input variables cannot be detected by observing high-integrity output variables. We remark that the notion of a low-integrity input variable can be naturally associated with a transmission error over a communications channel. Hence, we map the $i^{\mathtt{th}}$ possibly erroneous ciphertext symbol to a non-trusted low-integrity input (we are looking at the decryption algorithm that, in the case of RC4, is identical to the one used for encryption). The definition of non-interference can then conveniently be used to capture the absence of error propagation. For this, we associate the output plaintext symbols starting at position $i+1$ to trusted high-integrity outputs.

More precisely, our formulation captures the following idea: if an arbitrary change in the $i^{\mathtt{th}}$ input ciphertext symbol cannot be observed in the output plaintext symbols following position i, this implies that the stream cipher does not introduce error propagation in decryption.

3 Proofs by Composition

In this section we first review *self-composition*, a technique for proving non-interference based on deductive verification, and a generalisation (composition of two programs) that can be used to prove *program equivalences*. Our interest in reasoning about equivalence of programs is motivated by the notion of "correctness with respect to a reference implementation", as explained in Section 2.

The difficulties of applying self-composition in practice are well-known, and they also apply to proofs of equivalence. In Section 3.3 we will introduce the notion of *natural invariant* to overcome these difficulties. The technique establishes a correspondence between program annotations and an underlying formalisation of the operational semantics of the programs. This allows us to prove (interactively) certain fundamental lemmas that can be used to automatically prove properties based on the self-composition or composition techniques.

3.1 Self-Composition

The operational definition of non-interference involves two executions of the program but, using the *self-composition* technique [5], it can be reformulated to consider a single execution (of a transformed program). Given some (deterministic) 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. Non-interference can be formulated considering a single execution of the self-composed program $C;C^s$. Note that any state σ of $C;C^s$ can be partitioned into two states with disjoint domains $\sigma = \sigma^o \cup \sigma^s$ where $dom(\sigma^o) = Vars(C)$ and $dom(\sigma^s) = \{x^s | x \in Vars(C)\}$. C is information-flow secure if any terminating execution of the self-composed program $C;C^s$, starting from a state σ such that σ^o and σ^s differ only in the values of high-security variables, results in a final state σ' such that σ'^o and σ'^s are equivalent with respect to the values of low-security variables. This can be formulated without referring explicitly to the state partition: if $\sigma(x) = \sigma(x^s)$ for all $x \in V'_L$ and $(C;C^s,\sigma) \Downarrow \sigma'$, then $\sigma'(x) = \sigma'(x^s)$ for all $x \in V_L$.

Self-composition allows for a shift from an operational semantics-based to an axiomatic semantics-based definition, since the former can be written as the following Hoare logic partial correctness specification:

$$\left\{ \bigwedge_{x \in V'_L} x = x^s \right\} C;C^s \left\{ \bigwedge_{x \in V_L} x = x^s \right\}$$

Difficulties of Applying Self-composition. The example of Section 2.3 would result in the following self-composed program $F;F^s$.

```
f1 = 1; f2 = 0;
while (n > 0) {
  f1 = f1 + f2; f2 = f1 - f2; n--;
}
if (f1 > k) l = 1; else l = 0;
f1s = 1; f2s = 0;
while (ns > 0) {
```

```
  f1s = f1s + f2s; f2s = f1s - f2s; ns--;
}
if (f1s > ks) ls = 1; else ls = 0;
```

This example was used in previous work by Terauchi and Aiken [32] to show the difficulties of mechanising self-composition using software model checkers. In order to use a VCGen, one would annotate the self-composed code with the contract dictated by the following Hoare triple

$$\{n = n^s \wedge k = k^s \wedge l = l^s\}\ F; F^s\ \{l = l^s\}$$

together with the obvious control invariants for each loop (regarding the minimum value of the variables n and n^s).

Some of the generated proof-obligations would not however be discharged by an automatic prover. Admittedly, the control invariants do not sufficiently describe what the loops do (in particular, the fact that they are calculating Fibonacci numbers), and for this reason the post-condition cannot be proved, whether `n==ns` is included in the precondition (stating that n is not considered high-security) or not. The verification thus fails to recognize a secure program, even for such an apparently trivial example.

3.2 Equivalence by Composition

The above method can be extended to handle program equivalence. Suppose we have two programs C_1 and C_2, and that we are interested in proving their equivalence. Let V be the set of variables occurring in both programs (we assume both use the same set of variables, otherwise we may let $V = Vars(C_1) \cap Vars(C_2)$). 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:

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

Weaker notions of equivalence can be handled by taking V to be a subset of $Vars(C_1) \cap Vars(C_2)$. In fact, we are not restricted to equivalence relations – arbitrary relations can be considered between the two partitions of the state:

$$[R_1(\sigma, \sigma^s)]\ C_1; C_2^s\ [R_2(\sigma, \sigma^s)]$$

where σ and σ^s denote the state partitions associated with C_1 and C_2^s respectively.

The verification of such assertions leads to similar difficulties to those already mentioned for self-composition: in general there is no means to relate the outcomes of both programs, and automatic verification fails. This was to be expected, since establishing program equivalence is in general as undecidable problem.

3.3 Natural Invariants

In both scenarios identified above, the most evident difficulty of carrying out the verification comes from the absence of appropriate loop invariants. Of course, after finding these loop invariants we still need to establish the intended properties (ideally, with reasonable levels of automation). In what follows we propose a general approach to this problem. In short, it consists of the following steps:

1. Extracting a specification of each program from its relational semantics. We focus on the critical point of the verification process, which is the construction of appropriate loop invariants, and propose to construct them automatically. The invariants we extract constitute the *natural* specification of each program, guaranteed to be satisfied by it. Each invariant is named and turned into an predicate, which is then used to annotate the corresponding loop in the source code.
2. Identifying and interactively proving additional facts involving the named invariant predicates. The critical observation is that such lemmas correspond to basic *refactoring steps* that are recurrently used in the development of cryptographic software. Their purpose is to relate the specifications of the composed programs, capturing the non-trivial parts of the proofs required for verification.
3. Augmenting the source file with the previous facts (written as lemmas), which have been justified once-and-for-all by interactive proofs. The availability of these lemmas will allow automatic provers to carry out the verification process, validating the verification conditions generated by a potentially large number of (self-) composition proofs.

We recall that we are primarily interested in tackling self-composition, as well as program equivalences when both programs share much of the underlying control structure. This makes it reasonable to assume that the user may easily guide the interactive verification process by providing hints regarding the exploited code refactorings. This will allow them to take advantage of the high degree of automation that can be deployed to handle the remaining parts of the verification process.

Relational Specification. For concreteness, we consider a simple *While* language with integer expressions and arrays. Its syntax is given by:

$$
\begin{aligned}
P &::= \{P\} \mid \textbf{skip} \mid P_1; P_2 \\
&\mid V := E_{int} \mid A[E_{int}] := E_{int} \\
&\mid \textbf{if}\ (E_{bool})\ \textbf{then}\ P1\ \textbf{else}\ P2 \\
&\mid \textbf{while}\ (E_{bool})\ P \\
E_{int} &::= Const_{int} \mid E_{int}\ \text{op}\ E_{int} \mid A[E_{int}] \qquad \text{op} \in \{+,-,*,/,...\} \\
E_{bool} &::= true \mid false \mid \neg E_{bool} \mid E_{bool} \wedge E_{bool} \mid E_{bool} \vee E_{bool} \\
&\mid E_{int}\ \text{opRel}\ E_{int} \qquad \text{opRel} \in \{=,<,>,...\}
\end{aligned}
$$

We do not adopt any form of variable declaration. Instead, we consider a fixed `State` type that keeps 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). Array operations are axiomatised as usual:

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

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

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

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

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

The relation $\mathsf{loop}^n_{C,R}(\sigma,\sigma')$ denotes the loop specification for the body R under condition C. We call such a relation the *natural invariant* for the loop (strictly speaking, this is in fact a relation that provides a natural choice for a loop's invariant). 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. Nevertheless, 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.

Expressiveness and Relative Completeness. Natural invariants capture the input-output relational semantics of programs at the logical level. Naturally, they depend on a sufficiently expressive assertion language, as it should allow for the definition of new inductive relations. This corresponds essentially to Cook's expressiveness criteria in his relative completeness result for Hoare Logic [12]. In fact, from natural invariants we can easily recover the *strongest liberal predicate* as

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

An immediate consequence is that the verification of an arbitrary Hoare triple could be conducted logically, as follows

$$\begin{aligned}
\{P\}S\{Q\} \quad &\text{iff} \quad \mathrm{slp}(S,P) \supseteq Q \\
&\text{iff} \quad \forall\sigma\sigma',\ P(\sigma) \wedge \mathsf{spec}_S(\sigma,\sigma') \Rightarrow Q(\sigma').
\end{aligned}$$

However, we note that the presence of loops immediately forces the use of full-fledged inductive reasoning, compromising the aim of relying on automatic provers to conduct significant parts of the proof. We will thus confine such a general use of induction to general lemmas that will justify specific program transformations (refactorings).

Verifying Trivial Equivalences. Let us focus for a moment on the verification of the trivial equivalence by self-composition (any program is equivalent to itself). By construction, spec enjoys the following properties.

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

loop synchronisation: $\forall n_1\ n_2\ \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2'$,

$$\sigma_1 \equiv \sigma_2 \wedge \mathsf{loop}^{n_1}_{C,R}(\sigma_1,\sigma_1') \wedge \neg[\![C]\!](\sigma_1') \wedge \mathsf{loop}^{n_2}_{C,R}(\sigma_2,\sigma_2') \wedge \neg[\![C]\!](\sigma_2') \Longrightarrow n_1 = n_2;$$

loop determinism: $\forall n\ \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2'$,

$$\sigma_1 \equiv \sigma_2 \wedge \mathsf{loop}^{n}_{C,R}(\sigma,\sigma_1') \wedge \mathsf{loop}^{n}_{C,R}(\sigma,\sigma_2') \Longrightarrow \sigma_1' \equiv \sigma_2'.$$

Proof Both statements are proved by a simple induction (on $\max(n_1,n_2)$ in the first case, and n in the second). □

Proposition 1 *For every program fragment P and states σ_1, $\sigma_2, \sigma_1', \sigma_2'$,*

- spec *is a morphism that preserves $\equiv$. More precisely, if $\sigma_1 \equiv \sigma_2$, $\sigma_1' \equiv \sigma_2'$ and $\mathsf{spec}_P(\sigma_1,\sigma_1')$ then $\mathsf{spec}_P(\sigma_2,\sigma_2')$.*
- spec *is deterministic. More precisely, if $\mathsf{spec}_P(\sigma,\sigma_1')$ and $\mathsf{spec}_P(\sigma,\sigma_2')$ then $\sigma_1' \equiv \sigma_2'$.*

Proof By induction on the structure of P using Lemma 1.

Lemma 1 enables a fully automatic proof of equivalence of the self-composed program: it can be proved once-and-for-all and then included in the annotations provided to the verification platform, allowing all other proof obligations to be discharged. Indeed, our strategy for reasoning about multiple executions of the same (or related) program(s) is based on this observation: it is possible to identify a set of general lemmas that can be proven once-and-for-all, and that allow us to reason about self-composition assertions or to justify interesting refactorings (e.g. loop refactorings).

Self-composition Lemmas. The determinism property is not relevant to reason about a non-interference property by self-composition: it merely states that the two instances of the program will produce the same outputs when all of their inputs are equal. What is needed is a rephrasing of that property using an equality relation on low-security variables. If the control structure of the program does not depend on high-security variables, the determinism property proof can be carried over to non-interference lemmas. More explicitly, we recast each loop synchronisation lemma as follows

$$\begin{aligned}&\forall n_1\ n_2\ \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2',\\ &\quad \pi^C(\sigma_1) \equiv \pi^C(\sigma_2) \wedge \mathsf{loop}^{n_1}_{C,R}(\sigma_1,\sigma_1')\\ &\qquad \wedge \neg[\![C]\!](\sigma_1') \wedge \mathsf{loop}^{n_2}_{C,R}(\sigma_2,\sigma_2') \wedge \neg[\![C]\!](\sigma_2') \Longrightarrow n_1 = n_2\end{aligned}$$

where π^C projects the fragment of the state that influences the control state (i.e. the loop conditions) – note that this can be obtained by a simple (syntactical) dependency analysis that collects all variables accessed by C and all variables that may interfere on the values of the latter through the loop body. Then, a non-interference result for each loop follows easily from non-interference in its body:

$$\begin{aligned}&(\forall \sigma_1,\sigma_2,\sigma_1',\sigma_2',\ \ \sigma_1 \equiv_L \sigma_2 \wedge R(\sigma_1,\sigma_1') \wedge R(\sigma_2,\sigma_2') \Rightarrow \sigma_1' \equiv_L \sigma_2')\\ &\quad \Rightarrow \quad \forall \sigma_1,\sigma_2,\sigma_1',\sigma_2',\ \ \sigma_1 \equiv_L \sigma_2 \wedge \mathsf{loop}^{n_1}_{C,R}(\sigma_1,\sigma_1') \wedge \neg[\![C]\!](\sigma_1')\\ &\qquad \wedge \mathsf{loop}^{n_2}_{C,R}(\sigma_2,\sigma_2') \wedge \neg[\![C]\!](\sigma_2') \Rightarrow \sigma_1' \equiv_L \sigma_2'\end{aligned}$$

We observe that proving non-interference for loop-free programs by self-composition can be easily automated. The precondition for this lemma can be seen as an additional proof-obligation that must be verified.

Justifying Loop Refactorings. The main difficulty of justifying code refactorings comes up when the refactorings affect loops. 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 $\mathsf{loop}_{C,\mathsf{spec}(P)}(\sigma,\sigma')$ where P contains no loops. This case is sufficient to cover the program refactorings needed for establishing correctness of the RC4 `openSSL` implementation addressed in Section 5.

The simplest loop refactoring that can be addressed using our technique is *loop unrolling*, in which we detach instances of the loop body. This sort of transformation is justified by the following property that results from direct inversion of the definition of loop:

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

or, in iterated form:

$$\begin{aligned}&\forall n\ n'\ \sigma\ \sigma',\\ &\quad \mathsf{loop}^{n}(\sigma,\sigma'1) \wedge n' < n \Longrightarrow \exists\sigma'',\ \mathsf{loop}^{n'}(\sigma,\sigma'') \wedge \mathsf{loop}^{n-n'}(\sigma'',\sigma').\end{aligned}$$

Simple transformations like these are in fact better handled directly at the annotation level, rather than through explicit lemmas. Let us illustrate this by a small example that mimics an optimising transformation for the real-world example presented in Section 5. Consider the program

```
i := 0;
while (i<N) {
 x := x + y;
 i := i + 1
}
```

To implement it, the programmer chooses to unfold two copies of the original loop body in each iteration, yielding

```
N2 := N/2;
i := 0;
if (i<N2) then {
  while (i<N2) {
    x := x + y;
    x := x + y;
    i := i + 1
  };
  if (2*N2 <> N) then x := x + y else skip
}
else skip
```

To verify the equivalence between this implementation and the original program it suffices to identify the second loop invariant in the second program as the following,

$$\mathsf{loop}^{2*i}(\backslash\mathtt{old}(x,y,i*2),(x,y,i*2))$$

where $\backslash\mathtt{old}(x)$ evaluates x in the pre-state of the loop, and $\mathsf{loop}(-)$ refers to the natural invariant of the loop in the first program. By providing the invariant, we are making explicit the correspondence between both loop executions. This kind of guidance is reasonable to expect from someone intending to prove correctness of the target implementation.

Alternatively, one could establish that both programs are equivalent using direct logical arguments, as will now be explained. This would be the only option for more complex refactorings.

General Loop Fusions. To justify more significant code refactorings such as loop fusions (i.e. combining the bodies of two consecutive loops with the same control structure), we need to rely on an explicit lemma. Consider the equivalence between two consecutive loops (loops 1 and 2) and one single *fused* loop (loop 3). This is reminiscent of another real-world code refactoring that will occur in our case-study in Section 5.

Let us denote the natural invariants of these loops by $\mathsf{loop}_1, \mathsf{loop}_2$ and loop_3, respectively. Since we assume that

all the loops share the same control structure (loop condition and associated state), it is possible to prove *mixed* synchronisation lemmas such as

$$\forall n_1\ n_2\ \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2', \\ \pi^C(\sigma_1) \equiv \pi^C(\sigma_2) \wedge \mathsf{loop}_1^{n_1}(\sigma_1, \sigma_1') \wedge \neg [\![C]\!](\sigma_1') \\ \wedge \mathsf{loop}_2^{n_2}(\sigma_2, \sigma_2') \wedge \neg [\![C]\!](\sigma_2') \Longrightarrow n_1 = n_2.$$

The proof is a straightforward generalisation of the single loop version. Once this result has been established, one can prove the following main lemma that can be used to justify the fusion refactoring:

$$\forall n\ \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_1''\ \sigma_2', \\ \mathrm{BodyFusion}(\mathsf{body}_1, \mathsf{body}_2, \mathsf{body}_3) \wedge \mathrm{BodySwap}(\mathsf{body}_1, \mathsf{body}_2) \Rightarrow \\ \sigma_1 \equiv \sigma_2 \wedge \mathsf{loop}_1^n(\sigma_1, \sigma_1'') \wedge \mathsf{loop}_2^n(\sigma_1'', \sigma_1') \wedge \mathsf{loop}_3^n(\sigma_2, \sigma_2') \\ \Longrightarrow \sigma_1' \equiv \sigma_2',$$

where

$$\mathrm{BodyFusion}(\mathrm{R}_1, \mathrm{R}_2, \mathrm{R}_3) = \forall k,\ R_3^k \equiv (R_2^k \circ R_1^k) \\ \mathrm{BodySwap}(R_1, R_2) = \forall k k',\ k' < k \Rightarrow (R_2^k \circ R_1^{k'}) \equiv (R_1^{k'} \circ R_2^k)$$

BodyFusion and BodySwap denote simple properties concerning the loop bodies which, as was the case with the self-composition lemmas, are all non-recursive and can thus be regarded as additional proof-obligations, easily discharged by automatic provers.

4 Verification Infrastructure

In this work, we have used `Frama-c` [6], a tool for the static analysis of `C` programs that contains a multi-prover verification condition generator [18]. We also employed a set of proof tools that included the Coq proof assistant [33], and the `Simplify` [15], `Alt-Ergo` [11], and `Z3` [13] automatic theorem provers. `C` programs are annotated using the ANSI-C Specification Language (ACSL [6]). Both `Frama-c` and ACSL are work in progress; we have used the 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. Like axioms, lemmas can be used to prove goals, but unlike axioms, which require no proof, lemmas originate themselves new goals. In the proofs we developed, it was often the case that once an appropriate lemma was provided (and proved interactively with Coq), all the verification conditions could be automatically discharged.

In this section we describe our use of these tools to support the approach proposed in Section 3.

4.1 Specification Generation

The first step is to extract a relational specification from the program code. This process proceeds by recursion on the program structure (Section 3.3) and produces the specification as a logical formula.

In practice, it is convenient to produce the intended formula in prenex-form. This is easily accommodated introducing new fresh state variables in each elementary statement:

$$\mathsf{spec}_{S_1;S_2;\ldots;S_n}(x_1, x_2) = \exists w_0 \ldots w_n, \\ w_0 = x_1 \wedge w_1 = P_{S_1}(w_0) \wedge \ldots \wedge w_n = P_{S_n}(w_{n-1}) \wedge x_2 = w_n$$

where $P_S(\sigma)$ is the atomic state transformation associated with statement S. The extracted specification is then used according to the proof goal factoring method described in the previous section:

1. It is encoded in the Coq proof assistant to provide the context in which the required specific lemmas can subsequently be proved interactively;
2. It is included as ACSL loop invariant annotations in the C source code, to be fed to the `Frama-c` VCGen. For each loop specification the corresponding invariant is included in the ACSL code. In this step the lemmas proved in step 1 are also provided as ACSL lemmas in the annotated code, which should allow the remaining proof goals to be discharged automatically.

In the case study presented in Section 5 the specification was extracted by hand (see Appendix B), but we remark that the process is certainly amenable to mechanisation. Such a specification extraction tool will be developed for the domain-specific crypto language CAO [17] as a deliverable of the CACE project.

The Coq proofs mentioned in the first step above are constructed with support from a library that will now be described. Section 4.3 describes the second step above in more detail.

4.2 Coq Library

A Coq library was developed to support the proof of lemmas such as those introduced in Section 3. The library consists of several layers:

- *Frama-c interface*, which includes the logical theory exported by Frama-c and basic definitions/facts for reasoning with the theory inside Coq;
- *Basic loop support*, for basic treatment of loops (derivation of determinism and synchronisation lemmas);
- *Refactoring lemmas*: derivation of self-composition lemmas and loop-fusion lemmas;
- *Demos and applications*, which includes the RC4 example discussed in Section 5.

We have made extensive use of Coq's module system [9] in order to structure the development. As a rule, we embed each lemma and respective proof in a functor parameterised by basic facts it depends on. Concretely, we have defined the following.

- BuildLoopFun: functor that builds the inductive definitions for loops and derives the corresponding determinism lemmas. It is parameterised by two modules describing the loop state (the portion that affects the loop condition and its complement) and the specification of the loop body. These modules define the intended extensional equivalence on states and assert the determinism of the loop-body relation.
- BuildSyncFun: functor that establishes the synchronisation of two loops that share the same boolean condition. It is used, in particular, to derive a self-synchronisation lemma for each loop.
- BuildSelfComp: generates and proves the self-composition lemma. It is parameterised by the self-composition property and the proof that the loop body satisfies that property.
- BuildFusionFun: generates and proves the fusion lemma for two loops. This accepts the description of three loops (the loops to be fused and the resulting loop) together with the following properties:
  - *body-fusion property* – asserts that the body of the third loop behaves as the composition of the bodies of the first two loops;
  - *body-shift property* – asserts that iteration k of the first loop commutes with any of the first $k-1$ iterations of the second loop.

Note that all the results needed as inputs for the functors are non-recursive (they concern the loop body only) and can be expected to be proved successfully by an automatic prover.

4.3 Frama-c Usage

Frama-c takes as input annotated C programs in the form of ACSL files. In particular, loop invariants are mandatory for the verification to succeed. This means that in order to verify a property by composition, it is not enough to properly construct the composed program and to specify the intended contract (pre- and post-conditions) – this would certainly generate unprovable verification conditions. It is also required to complement the ACSL file with definitions and annotations. The following steps detail the procedure needed to perform the verification:

1. Including ACSL definitions corresponding to the inductive properties associated to each loop (see Section 4.1);
2. For each loop specification, annotating the program with a loop-invariant of the form

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

   where C and R are the loop's condition and body, and σ@Init denotes the snapshot of the loop's initial state (Frama-c supports this notion through the use of explicit state labels in annotations).
3. Augmenting the ACSL file with specific lemmas (proved in Coq with the support of the library of Section 4.2);
4. Generating proof obligations with Frama-c;
5. Using an automatic prover (e.g. Simplify) to discard the generated obligations.

The choice of the required lemma is based on the specific property under scrutiny (e.g. a self-composition lemma for a non-interference property). We remark that this user-dependent choice is an important ingredient for the success of the verification process. The goal of our method is to allow the user to concentrate on this critical part of the verification process by providing assistance in dealing with the remaining tasks, which are tedious but luckily prone to automation.

5 Case Study: openSSL implementation of RC4

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 [31]. RC4 is widely used in commercial products, as it is included as one of the recommended encryption schemes in standards such as TLS, WEP 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 sequence k_t that approximates a perfectly

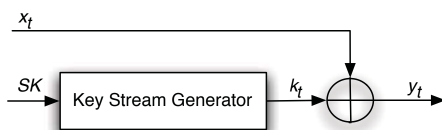

Fig. 1 Block diagram of the RC4 cipher

random bit string, and is independent of plaintext and ciphertext (we adopt the most widely used version of RC4, implemented in `openSSL`, which operates over byte-sized words). The encryption operation consists simply of XORing each plaintext byte x_t with a fresh key stream 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 immaterial 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`). The same function can be used for decryption by providing the ciphertext in the `indata` buffer. 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. In the rest of this section we briefly present the verification activities performed on this case-study. Full details of it, including all the annotated source files used, can be found in [1].

5.1 Verification of Safety Properties

The `Frama-c` VCGen allows users to perform a safety analysis of C code, which may be run independently of functional verification (see Section 2.2). This produces a special class of verification conditions (called *safety conditions*) that are not generated from contracts. Their validity implies that the program will execute safely with respect to a restricted set of common programming errors that may result in incorrect or unreliable implementations, or even with respect to security vulnerabilities. These comprise memory safety, including the absence of buffer overflows, and also absence of numeric errors due to overflows in integer calculations.

Note that, even though the proof obligations generated for the safety analysis do not result from explicit assertions made by the programmer, it is usually necessary to annotate the code with preconditions that permit justifying the proof goals. These preconditions limit the analysis to function executions for which the caller has provided valid inputs. For example, in RC4 one must assume that the `indata` and `outdata` arrays have a valid addressable range between `0` and `len-1` for the safety conditions to be valid. The inclusion of simple loop invariants to enable reasoning about the program state before, during and after each loop execution is also required. Finally, and given that cryptographic code tends to make use of some arithmetic operators that are not commonly used in other application domains, we noted that `Frama-c` lacked appropriate support in some cases, namely for bit-wise operators. To overcome this difficulty we added some very simple axioms to the annotated RC4 code.

Running the `Frama-c` VCGen on the annotated source code RC4 gave rise to 869 verification conditions. All of these were automatically discharged using a set of automatic provers that included `Simplify`, `Alt-Ergo`, and `Z3`.

5.2 Error-propagation Property

We have used the self-composition technique described in Section 2.3 to verify whether the RC4 implementation in `openSSL` indeed satisfies a property which is common to all synchronous stream ciphers: the absence of error propagation. Recall that this amounts to verifying that an erroneous (possibly tampered) input symbol, which will unavoidably result in a corresponding erroneous output symbol in the same position, will not affect subsequent outputs. Formally, following the notation introduced in Section 2.3 that associates V_H with the set of low-integrity input variables and V_L with the set of high-integrity outputs, we have for some $i \in [0, len[$:

$$\begin{aligned} V_H &= \{\texttt{indata}[i]\}, \\ V_L &= \{\texttt{outdata}[j] \mid i < j < \texttt{len}\}. \end{aligned}$$

We have extracted natural invariants from the code and annotated the source file according to the procedure presented in Section 4. The verification with `Frama-c` resulted in the generation of 17 proof obligations, all of which were

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

  x=key->x; y=key->y; d=key->data;
  x=((x+1)&0xff);      tx=d[x];
  y=(tx+y)&0xff;       d[x]=ty=d[y];
  d[y]=tx;  key->x=x; key->y=y;
  return d[(tx+ty)&0xff];
}

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 reference implementation

```
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

automatically discharged by `Simplify`. This was made possible by the inclusion of a helper lemma in the ACSL annotations (proved offline in Coq by instantiating the appropriate functor from the developed library).

5.3 Correctness of RC4 Openssl Implementation

A direct transcription to a C implementation of the RC4 specification presented at the beginning of this section 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 implementations.

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 to speed 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 synchronous ciphers such as RC4, the number of bits in the key stream that can be pre-computed can be arbitrarily large, as this is totally independent of the encrypted data. The version of RC4 shown in Figure 3 moves in this direction by separating the key stream generation process from the plaintext masking (or ciphertext unmasking) process. This is an instance of the loop-fusion refactoring of Section 3.3. The infrastructure described in Section 4 was used to prove equivalence beween the programs in Figures 2 and 3. Appendix B shows an example of how the deductive verification tool is interfaced in this case.

A sequence of refactorings leading to the `openssl` *implementation.* We 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: (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 equivalent bit-wise operations. A macro is also introduced to improve readability.

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.

```
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 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. Equivalence checking for these low-level refactorings was performed directly in `Frama-c`.

6 Related Work

A good survey of language-based information flow security can be found in [30]. A good general view of self-composition can be found in [10]. Information flow policies were first introduced by Denning et. al [14] and tend to be formalised as noninterference properties. Information flow type systems, have been used to enforce noninterference in different contexts [36,27,26,34,35]. The main challenge in designing these systems is that they are often too conservative in practice, so that secure programs may be rejected. Leino and Joshi [24] were the first to propose a semantic approach to checking secure information flow, with several desirable features: a more precise characterisation of security; it applies to all programming constructs whose semantics are well-defined; and it can be used to reason about indirect information leakage through variations in program behaviour (e.g., whether or not the program terminates). An attempt to capture this property in program logics using the *Java Modelling Language* (JML) [23] was presented by Warnier et al. [37], who proposed an algorithm, based on the strongest postcondition calculus, that generates an annotated source file with specification patterns for confidentiality in JML. Dufay et al. [16] have proposed an extension to JML to enforce non-interference through self-composition. This extended annotation language allows for a simple definition of non-interference for Java programs. However, the generated proof obligations are complex, which limits the general applicability of the approach.

Terauchi and Aiken [32] identified problems in the self-composition approach, arguing that automatic tools (software model checkers like SLAM [2] and BLAST [19]) are not powerful enough to verify this property over programs of realistic size. To compensate for this, the authors propose a program transformation technique for an extended version of the self-composition approach. Rather than replicating the original code, the renamed version is interleaved and partially merged with it. Naumann[28] extended Terauchi and Aiken's work to encompass heap objects, presented a systematic method to validate the transformations proposed in [32], and reported on the experience of using these techniques with the Spec# [4] and ESC/JAVA2 [21] tools.

Natural Invariants provide an explicit rendition of program semantics. In [25] 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 strengths of higher-order logic reasoning with facilities provided by automatic first-order provers.

Relational Hoare Logic [7] has 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.

7 Conclusion

We have used an off-the-shelf verification platform to check several classes of properties 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. Our results focus on three security-relevant properties of this implementation, with increasing degrees of verification complexity: (1) safety properties such as the absence of numeric errors and memory safety; (2) absence of error propagation formalised as non-interference; and (3) functional equivalence with respect to a reference implementation.

In more concrete terms, we have used Frama-c to prove that the RC4 implementation does not cause null pointer dereferencing exceptions, and always performs array accesses with valid indices. In other words, the implementation is secure against *buffer overflow* attacks. Additionally, we demonstrated that the limited ranges of numeric variables used in the RC4 implementation are guaranteed not to introduce calculation errors for particular input values.

An important property of stream ciphers such as RC4 is their behaviour when a bit in the ciphertext is flipped over a communication channel. The behaviour of RC4 is common to other *synchronous* ciphers: bit errors are not propagated in any way, i.e. if a ciphertext bit is flipped during transmission, then only the corresponding plaintext bit is affected. We have formalised this property as a novel application of the *non-interference* concept, widely used in the formalisation and verification of secure information flow properties, and subsequently proved that the RC4 implementation indeed enjoys this property.

Finally, we have also shown how the method introduced to prove non-interference can be applied to the more general case of *equivalence proofs*, to prove the correctness of real implementations with respect to reference implementations. Cryptography is a prime candidate for equivalence proofs, since specifications are usually given as reference implementations rather then using some high level model or language. In concrete terms we have proved the equivalence between a reference implementation of RC4 and the realistic implementation included in openSSL.

Program equivalences are difficult verification challenges by nature, and automatic proof is of little help. Resorting to an interactive proof tool to conduct inductive proofs involving loops is inevitable. Our approach can be summed up as follows

1. Program equivalences in general can be expressed 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. Natural invariants are good candidates for establishing the connection between the interface specification language and the proof assistant: the ACSL specification language admits inductively defined predicates, thus the natural invariants annotated into the specification files (fed to the VCGen) can make use of them, and lemmas can also be included in these files, to be (i) used by automatic provers and (ii) exported to Coq for interactive proof. These predicates / invariants (and some standard lemmas) can be generated mechanically. Note that since the typical first-order prover does not support inductive predicates, Frama-c will replace them by uninterpreted predicates in the verification conditions generated for these provers (with axioms corresponding to the interactively proved properties). This allows to capture the program semantics through purely first-order assertions.
3. Concluding the verification process is then a matter of establishing and proving interactively a small number of adequate lemmas that concentrate the more creative parts of the proofs required in the verification process. To assist users in this more demanding task, we have developed a dedicated Coq library. Once a lemma has been proved in Coq and annotated into the composed specification of the refactoring step, all the proof obligations generated by Frama-c are discharged automatically.

In addition to showing that deductive verification methods are increasingly more amenable to practical use with reasonable degrees of automation, our work answers some open questions raised by previous work, which seemed to indicate that proofs by (self-)composition were not directly applicable in real-world situations. Our results are promising in that we have been able to achieve our goal using only off-the-shelf verification tools.

What is more, we believe that our technique has a high potential for mechanisation. For instance, it is likely that the procedure referred in Section 4.3 may itself be partially automated. Speculating a bit, it is conceivable to extend ACSL with constructs that mechanise some of the steps:

```
\\@ lemma self_comp_lemma : \SelfCompLemma(loop1, ...);
...
loop1:
\\@ loop invariant \natural_invariant(loop1);
...
```

An advantage of such an integration is that these annotations might trigger the generation of auxiliary proof obligations, such as those required by the functor that generates the self-composition lemma.

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

References

1. José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Deductive verification of cryptographic software. Technical Report DI-CCTC-09-03, CCTC, Univ. Minho, 2009. Available from http://cctc.uminho.pt/publications?year=2009.
2. Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.
3. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005.
4. Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.
5. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE Computer Society, 2004.
6. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specfication Language*. CEA LIST and INRIA, 2008. Preliminary design (version 1.4, December 12, 2008).
7. Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 14–25. ACM, 2004.
8. Computer Aided Cryptography Engineering. EU FP7. http://www.cace-project.eu/.
9. Jacek Chrzaszcz. Implementation of modules in the Coq system. In David Basin and Burkhart Wolff, editors, *Proceedings of the Theorem Proving in Higher Order Logics 16th International Conference*, volume 2758 of *LNCS*, pages 270–286, Rome, Italy, September 2003. Springer.
10. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *CSF*, pages 51–65. IEEE Computer Society, 2008.
11. Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.
12. Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.
13. Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.
14. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
15. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
16. Guillaume Dufay, Amy Felty, and Stan Matwin. Privacy-sensitive information flow with JML. In *Automated Deduction - CADE-20*, pages 116–130. Springer Berlin / Heidelberg, August 2005.
17. Dan Page (editor). CACE Deliverable D1.1: Complete CAO and qhasm specifications, 2009. Available from http://www.cace-project.eu.
18. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
19. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
21. B. P. F. Jacobs, J. R. Kiniry, M. E. Warnier, Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In *FMCO 2002: Formal Methods for Component Objects, Proceedings, volume 2852 of Lecture Notes in Computer Science*, pages 202–219. Springer, 2003.
22. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):1–54, 2009.
23. Gary T. Leavens, Clyde Ruby, K. Rustan M. Leino, Erik Poll, and Bart Jacobs. JML (poster session): notations and tools supporting detailed design in Java. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 105–106, New York, NY, USA, 2000. ACM.
24. K. Rustan M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Lecture Notes in Computer Science*, 1422:254–271, 1998.
25. Daniel Leivant. Logical and mathematical reasoning about imperative programs. In *POPL*, pages 132–140, 1985.
26. Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.
27. Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.
28. David A. Naumann. From coupling relations to mated invariants for checking information flow. In *Computer Security - ESORICS 2006*, volume 4189 of LNCS, pages 279– 296, 2006.
29. The OpenSSL Project. http://www.openssl.org.
30. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
31. Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C.* Wiley, New York, 2nd edition, 1996.
32. Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.
33. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. http://coq.inria.fr.
34. Stephen Tse and Steve Zdancewic. A design for a security-typed language with certificate-based declassification. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2005.
35. Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206. IEEE Computer Society, 2007.
36. Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer, 1997.
37. Martijn Warnier and Martijn Oostdijk. Non-interference in JML. Technical Report ICIS-R05034, Nijmegen Institute for Computing and Information Sciences, 2005.

A openSSL implementation of RC4

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

void RC4(RC4_KEY *key,const unsigned long len,
    unsigned char *indata,
  unsigned char *outdata)
{
         register unsigned char *d;
         register unsigned char 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;
}
```

B ACSL loop specification

```
/*@ predicate eqAk{L1,L2}(integer k,
  @                      unsigned char u1[],
  @                      unsigned char u2[]) =
  @  \forall integer l;
  @    l!=k ==> \at(u1[l],L1)==\at(u2[l],L2);
  @*/

/*@ predicate eqA{L1,L2}(unsigned char u1[],
  @                     unsigned char u2[]) =
  @  \forall integer l; \at(u1[l],L1)==\at(u2[l],L2);
  @*/

/*@ predicate RC4NextKeySymbol{L1,L2}(unsigned char *x,
  @                                  unsigned char *y,
  @                                  unsigned char d[],
  @                                  unsigned char k) =
  @  \exists unsigned char tx, unsigned char ty;
  @   \at(*x,L2) == ((\at(*x,L1) + 1) & 0xff) ==>
  @   tx == \at(d[\at(*x,L2)],L1) ==>
  @   \at(*y,L2) == ((tx+\at(*y,L1)) & 0xff) ==>
  @   ty == \at(d[\at(*y,L2)],L1) ==>
  @   \at(d[\at(*x,L2)],L2) == ty ==>
  @   \at(d[\at(*y,L2)],L2) == tx ==>
  @   k == (\at(d[(tx+ty)&0xff],L1);
  @*/

/*
 *     spec1{L,Here}((int)0,i,key,&(x),&(y),d);
 *     Invariant for the first loop in  Fig.3.
 */

/*@
  @ inductive spec1{L1,L2}(integer i1,integer i2,
  @                       unsigned char key[],
  @                       unsigned char *x,
  @                       unsigned char *y,
  @                       unsigned char d[]) {
  @  case spec1_base{L} :
  @    \forall integer i1,integer i2,
  @            unsigned char key[],
  @            unsigned char *x,
  @            unsigned char *y,
  @            unsigned char d[];
  @     i1 == i2 ==> spec1{L,L}(i1,i2,key,x,y,d);
  @ case spec1_step{L1,L2,L3} :
  @    \forall integer i1,integer i2,integer i3,
  @            unsigned char key[],
  @            unsigned char *x,
  @            unsigned char *y,
  @            unsigned char d[],
  @            unsigned char k;
  @    spec1{L1,L2}(i1,i2,key,x,y,d) ==>
  @    eqAk{L2,L3}(i2,key,key) ==>
  @    RCNextSymbolKey{L2,L3}(x,y,d,k) ==>
  @    \at(key[i2],L3) == k ==>
  @    i3 == i2 +1 ==>
  @    spec1{L1,L3}(i1,i3,key,x,y,d);
  @ }
  @*/
```