# Deductive Verification of Cryptographic Software

José Bacelar Almeida Manuel Barbosa Jorge Sousa Pinto
Bárbara Vieira

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

**Abstract**

We report on the application of an off-the-shelf verification platform to the RC4 stream cipher cryptographic software implementation (as available in the openSSL library), and introduce a deductive verification technique based on self-composition for proving the absence of error propagation.

## 1 Introduction

Software implementations of cryptographic algorithms and protocols are at the core of security functionality in many IT products. They are also vital components in safety-critical systems, namely in the military area. However, the development of this class of software products is understudied as a domain-specific niche in software engineering. This development is clearly distinct from other areas of software engineering due to a combination of factors. Firstly, cryptography is an inherently inter-disciplinary subject. The design and implementation of cryptographic software draws on skills from mathematics, computer science and electrical engineering. Such a rich body of research is difficult to absorb and apply without error, even for 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 focused on desktop class processors within computers in our offices and homes. 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 optimise each algorithm with respect to each platform.

CACE (Computer Aided Cryptography Engineering, `http://www.cace-project.eu`) is an European Project that targets the lack of support currently offered to cryptographic software engineers. The central objective is the development of a tool-box of domain-specific languages, compilers and libraries, that supports the production of high quality cryptographic software. 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 validate security-relevant properties of a `C` implementation of the `RC4` encryption scheme that is included in the well-known open-source library `openSSL` `http://www.openssl.org`). Additionally, we have developed a framework that permits automating most of the steps required to verify non-interference using the self-composition approach introduced in [3]. We believe this verification technique may be of independent interest.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 146-155

## 2 Background

**Deduction-based Program Verification.** The techniques employed in this paper are based on Hoare Logic [9], brought to practice through the use of *contracts* – specifications consisting of preconditions and post-conditions, 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 are 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.

The concrete tools we have used in this work were `Frama-c` [4], a tool for the static analysis of `C` programs annotated using the ANSI-C Specification Language (ACSL [4]). `Frama-c` contains a multi-prover VCGen [8], that we used together with a set of proof tools that included the Coq proof assistant [18], and the Simplify [6] and Ergo [5] automatic theorem provers. Both `Frama-c` and ACSL are very much work in progress; we have used the latest release of `Frama-c` (known as Lithium).

A few features of the `Frama-c` VCGen should be emphasized in the context of our work. The first is the possibility to export individual proof obligations to different proof tools, which allows users to first try discharging them with one or more automatic provers, leaving the harder conditions to be exported to an interactive proof assistant. A second feature is the declaration of lemmata: results that can be used to prove goals, but give themselves origin to new goals. We have used this possibility to clearly isolate the parts of proofs that cannot be handled automatically. It is often the case that the difficulty resides in a single result that has to be proved interactively, but whose existence as a lemma allows for all the remaining conditions to be proved automatically. The possibility to define inductive predicates (passed on to the proof tool as axiomatized predicates, and kept as inductive definitions if the tool supports them, as is the case of Coq), and the *labels* mechanism that allows the value of an expression in a program state to be used explicitly in logical expressions, are two additional features that we have made use of.

Finally, `Frama-c` generates *safety conditions* – special verification conditions (not generated from contracts), whose validity implies that the program will execute safely with respect to, say, memory accesses or integer overflow. Safety verification may be run independently of functional verification.

**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. A dual formulation permits capturing security policies which constrain information flow from non-trustworthy (or low-integrity) inputs, to trusted (or high-integrity) outputs. We provide a short overview of developments in this area related to the work in this paper in Section 6.

Consider the more common case of secure information-flow that aims to preserve data confidentiality. Information may flow from high-security to low-security either directly via assignment instructions, or indirectly. The following code from Terauchi and Aiken [17] computes in $f_1$ the $n^{\mathrm{th}}$ Fibonacci number and then assigns a value to $l$ that depends on the value of $f_1$.

```
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 $f_1$, and assignments to the latter variable are performed inside a loop that is controlled by the high security condition $n > 0$. The program is thus insecure. If $n$ were not high security, the program would 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 $f_1$. 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 $V_\uparrow$ and $V_\downarrow$ denote respectively the sets of high-security and low-security variables of $C$, and $V_\downarrow{}' = \mathit{Vars}(C) \setminus V_\uparrow$. We write $(C, \sigma) \Downarrow \tau$ to denote the fact that when executed in state $\sigma$, $C$ stops in state $\tau$ (i.e. $\Downarrow$ is the evaluation relation in a big-step semantics of the underlying language). Then the program $C$ is secure if $\sigma \overset{V_\downarrow{}'}{=} \tau \ \wedge \ (C, \sigma) \Downarrow \sigma' \ \wedge \ (C, \tau) \Downarrow \tau' \implies \sigma' \overset{V_\downarrow}{=} \tau'$ for arbitrary states $\sigma$, $\tau$, where $\sigma \overset{X}{=} \tau$ denotes the fact that $\sigma(x) = \tau(x)$ for all $x \in X$, i.e. $\sigma$ and $\tau$ are $X$-indistinguishable.

**Self-composition.** The operational definition of non-interference involves two executions of the program, but it can be reformulated so that a single execution (of a transformed program) is considered, using the *self-composition* technique [3]. 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 $x^s$. Non-interference can be formulated considering a single execution of the self-composed program $C;C^s$. Note that any state $\sigma$ of $C;C^s$ can be partitioned into two states with disjoint domains $\sigma = \sigma^o \cup \sigma^s$ where $\mathit{dom}(\sigma^o) = \mathit{Vars}(C)$ and $\mathit{dom}(\sigma^s) = \{x^s \mid x \in \mathit{Vars}(C)\}$. $C$ is information-flow secure if any terminating execution of the self-composed program $C;C^s$, starting from a state $\sigma$ such that $\sigma^o$ and $\sigma^s$ differ only in the values of high-security variables, results in a final state $\sigma'$ such that $\sigma'^o$ and $\sigma'^s$ 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 $x \in V_\downarrow{}'$ and $(C;C^s, \sigma) \Downarrow \sigma'$, then $\sigma(x) = \sigma(x^s)$ for all $x \in V_\downarrow$.

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 Hoare logic partial correctness specification $\{\bigwedge_{x \in V_\downarrow{}'} x = x^s\}\, C;C^s\, \{\bigwedge_{x \in V_\downarrow} x = x^s\}$ which can be verified, for instance, as an ACSL contract.

An ACSL annotated program can easily be written for this example. Note that ACSL contracts pertain to functions, and as such a C function must be created whose body is the self-composed program. The contract contains a precondition stating that initial values of all non-high security variables are pairwise equal (for both composed copies of the program), and a post-condition stating that the values of low security variables are pairwise equal. Running `Frama-c` on this code, with the obvious 'control' invariants for each loop (regarding the minimum value of the variables $n$ and $n^s$) also annotated, produces (with safety verification turned off) a total of 10 proof obligations, 2 of which cannot be automatically discharged. 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 (meaning that $n$ is not considered high-security) or not. The verification thus fails to recognize a secure program. The difficulties of applying self-composition in practice have been previously identified (see Section 6); in this paper we introduce a technique that can help in successfully applying it with the help of a deductive verification system.

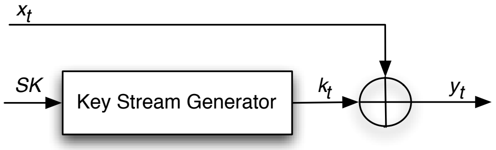

Figure 1: Block diagram of the RC4 cipher

## 3 RC4 and Verification of its Safety Properties 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 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 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 XOR-ing 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.

$$i_t = i_{t-1} + 1 \ , \ j_t = j_{t-1} + S_{t-1}[i_t] \ , \ S_t[i_t] = S_{t-1}[j_t] \ , \ S_t[j_t] = S_{t-1}[i_t] \ , \ k_t = S_t[S_t[i_t] + S_t[j_t]]$$

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 the full version of this paper [1] 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. We have added ACSL annotations to this RC4 implementation in order to facilitate the verification of a set of safety properties. These comprise memory safety, including the absence of buffer overflows, and also the absence of numeric errors due to overflows in integer calculations. This annotated version of RC4 was processed using Frama-c, which generated 869 verification conditions. All of these verification conditions could be automatically proved using tools such as Alt-Ergo and Z3. We highlight some important points in this verification work.

Frama-C interprets C primitive types (e.g. char, int, etc.) as integers with different precisions, based on the number of bits of each type. This means that a number of proof obligations must be automatically generated to ensure the validity of each arithmetic operation, by imposing range limits on the corresponding results. Proof obligations that ensure that every memory access is safe are also automatically generated. Note that, even though these proof obligations do not result from explicit assertions made by the programmer, it is usually necessary to annotate the code with preconditions. These 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 proof obligations to be valid. The required preconditions also include the validity of the memory region in which the RC4 key stream generator state (`key`) is passed to the function, and bounding the length of the input and output buffers (`len`).

The verification of the proof obligations generated by `Frama-C` also required the annotation of the RC4 code with loop invariants. These invariants are critical to enable the deductive verification process to reason about the program state before, during and after each loop execution. For example, in the first loop, the invariant permits establishing that index `i` lies between `0` and `len>>3L`, and also keeping track of how the `indata` and `outdata` pointer values change during the loop execution. Given the high number of proof obligations to be proved, and to guide the automatic provers in the process of establishing the validity of some of these conditions, additional assertions were introduced in the code. For example, at the end of the first loop, one assertion is introduced to *force* the provers to pinpoint the condition that must be valid at the end of the loop execution. 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 the proof tools 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 which express bounds on the outputs of these operators.

## 4 Towards Automating Proofs by Self-Composition.

Let us consider again the Fibonacci example of Section 2. The success of being able to prove automatically the security of programs using a Hoare logic-based tool such as `Frama-c` depends totally on adequately annotating the program. The ACSL program could be annotated with loop invariants describing the functional behaviour of each loop, which would require a formalisation of Fibonacci numbers. This kind of functional property would allow for the post-condition to be proved (both loops would be annotated as calculating Fibonacci numbers); however, such properties are hard to produce and not adequate for an automatic approach that should be applicable independently of what the code does.

A more feasible alternative is to use an abstract invariant that captures in logical form the state transformation associated with the loop, written as a formula that uses an inductively defined predicate (as supported by `Frama-c`). We will refer to these as the loop's *natural invariant*. Let $\vec{v}$ denote the vector of variables used inside a given loop. The predicate $\mathit{natinv}(\vec{v}_i, \vec{v})$ will be defined with the meaning that execution of the loop started with initial values of the variables given by $\vec{v}_i$; and the current values of the variables are given by $\vec{v}$. The inductive definition of this predicate has in general a base case of the form $\mathit{natinv}(\vec{v}_i, \vec{v}_i)$ (corresponding to the loop initialization) and an inductive case of the form $\mathit{natinv}(\vec{v}_i, \vec{v}_t) \rightarrow B \rightarrow R(\vec{v}_t, \vec{v}) \rightarrow \mathit{natinv}(\vec{v}_i, \vec{v})$, where $B$ is the boolean condition of the loop, and the formula $R(\vec{v}_t, \vec{v})$ relates the values of the variables in two successive iterations.

Turning back to our example, the predicate that gives rise to the natural invariant of the loop, written $\mathit{natinv}(f_{1i}, f_{2i}, n_i, f_1, f_2, n)$, is defined inductively by the following two cases:

$$\begin{array}{rl} \text{Base case:} & \forall f_{1i}, f_{2i}, n_i.\ \mathit{natinv}(f_{1i}, f_{2i}, n_i, f_{1i}, f_{2i}, n_i) \\ \text{Inductive case:} & \forall f_{1i}, f_{2i}, n_i, f_{1t}, f_{2t}, n_t, f_1, f_2, n.\ \mathit{natinv}(f_{1i}, f_{2i}, n_i, f_{1t}, f_{2t}, n_t) \rightarrow \\ & \quad n > 0 \rightarrow (f_1 = f_{1t} + f_{2t} \wedge f_2 = f_{1t} \wedge n = n_t - 1) \rightarrow \mathit{natinv}(f_{1i}, f_{2i}, n_i, f_1, f_2, n) \end{array}$$

The natural invariant of the loop is then written simply as $\mathit{natinv}(f_1@\mathbf{Init}, f_2@\mathbf{Init}, n@\mathbf{Init}, f_1, f_2, n)$, where $x$@**Init** denotes the value of $x$ in the initial state of the loop execution. We remark that this is quite an atypical use of loop invariants, since it relates the values of variables in different states. For such a small example the definition of the natural predicate can be written by hand, but the point here is that for more realistic programs a symbolic evaluation algorithm can be used to synthesize the natural invariants.

The ACSL self-composed program annotated with the natural invariant and the inductive predicate definition is included in the full version of this paper [1]. Of the 16 VCs generated by `Frama-c`, 2 cannot be discharged automatically, and require interactive proof.

**A General Framework.** The hard part in the proof above is the fact that the inductive definition is deterministic in the sense that in $\mathit{natinv}(f_{1i}, f_{2i}, n_i, f_1, f_2, n)$ the values of $f_1$ and $f_2$ are uniquely determined by the values of $f_{1i}$, $f_{2i}$, $n_i$, and $n$ (i.e. the current state of the loop is uniquely determined by the initial state and the current iteration). We isolate this as the following lemma:

$$\forall f_{1i}, f_{2i}, n_i, f_{1a}, f_{2a}, n_a, f_{1b}, f_{2b}, n_b.\ \mathit{natinv}(f_{1i}, f_{2i}, n_i, f_{1a}, f_{2a}, n_a) \rightarrow \mathit{natinv}(f_{1i}, f_{2i}, n_i, f_{1b}, f_{2b}, n_b) \rightarrow \\ n_a = n_b \rightarrow (f_{1a} = f_{1b} \wedge f_{2a} = f_{2b})$$

The lemma clearly implies that two executions of the loop starting from the same initial conditions are synchronized (in fact a weaker lemma would be sufficient for our purposes, since only the final state of the loop is relevant). Once this lemma is added to the ACSL specification, all the VCs are automatically discharged (the program is proved secure). Hoping for an automatic proof of the lemma would be too ambitious. However, we found that an interactive proof in Coq is straightforward and can be conveniently decomposed. Furthermore, such lemmata are fairly easy to write and can also be generated automatically. Based on this observation, we have developed a method to construct self-composition proofs that aims to maximise the degree of automation and essentially eliminate the need to interactively use the prover. For this, we isolate the small parts of the process that clearly require user intervention, allowing the user to just *fill-in the blanks*. In this direction, rather than stand-alone proofs for particular applications of self-composition, we have produced a Coq development that formalises natural invariants in general, and can be used to generate proofs of desired lemmata. This development is presented in the full version of this paper [1]. In a nutshell, a relational specification is first extracted from the annotated program, which is then completed with a small set of facts provided by the user. The catch is that, since these facts are kept very simple (non-inductive), they can be checked automatically. This specification is then used to instantiate Coq modules containing definitions and basic facts, from which Coq can then generate and automatically prove the desired lemma.

**Reasoning with Arrays.** In Section 5 we will employ the self-composition technique to the RC4 algorithm. In that context, some of the variables that must be checked to be equal at the end of the self-composed program execution are array variables (e.g. the high-integrity outputs). The appropriate notion of array equality that must be used in preconditions and post-conditions when applying the self-composition technique is of course *extensional*. Equality of array variables corresponds to equality of memory addresses, which is unsuitable to capture equality of inputs and outputs as required. Moreover, the renaming operation $C^s$ used by the self-composition technique must now allocate a new array for every array variable in the program $C$. Our Coq formalisation contemplates these aspects. An aspect of `Frama-c` that is required for natural invariants involving arrays is the possibility to have *state parameters* in predicates. The $\mathit{natinv}(\vec{v}_i, \vec{v})$ predicate was described as having as parameters the values of variables in two different states; for array variables a single array parameter is used, together with two state parameters. It is then possible to refer to the contents of array positions in those states. Having two array parameters would be wrong, since both would be passed the same memory address. Determinism lemmata as described above must also parametrised by states, and basically express properties like “if an execution of a loop starts in state $S_1$ and ends in $S_2$; a second execution of the same loop starts in state $S_3$ and ends in $S_4$; and moreover the contents of a given array is the same in states $S_1$ and $S_3$, then those contents will also be the same in states $S_2$ and $S_4$".

## 5 Using Information Flow to Capture Error Propagation

An important property of stream ciphers such as RC4 is how they behave when used to transfer data over channels which may introduce transmission errors. In particular, it is relevant how the decryption process reflects a wrong ciphertext symbol in the resulting plaintext: depending on the cipher 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 to select 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, and which are therefore more easily detected.

In this section, we show how a generic technique for verifying secure information flow in software implementations can be used to check that the RC4 implementation in `openSSL` does not introduce error propagation. Given that this property is a known feature of synchronous ciphers such as RC4, it is clear that a complete proof of functional correctness with respect to a reference specification (such as the one introduced in the next section) would imply this result. However, the idea here is to demonstrate that general purpose verification tools and techniques developed for other software domains can be usefully adapted to verify relevant (and to the best of our knowledge never before addressed in literature) security properties in cryptographic software implementations.

**Formalising Error Propagation as Non-Interference.** The goal is to capture the error propagation property using a well studied formalism, so that we can take advantage of existing verification techniques to check this property in stream cipher implementations. To do this, we recall the notion of non-interference that was introduced in Section 2 as a possible formalisation of secure information flow. The intuition underlying this formalisation is that secure information flow can be guaranteed by checking that arbitrary changes in high-security (or low-integrity) input variables cannot be detected by observing the values of low-security (or high-integrity) output variables. Observe that the notion of a low-integrity input variable can be naturally associated with that of 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 non-interference definition can then conveniently be used to naturally 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 other words, we want to verify 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 that associates $V_\uparrow$ with the set of low-integrity input variables and $V_\downarrow$ with the set of high-integrity outputs, we have for some $i \in [0, len[$:

$$V_\uparrow = \{\mathtt{indata}[i]\}, \quad (1)$$

$$V_\downarrow = \{\mathtt{outdata}[j] \,|\, i < j < \mathtt{len}\}. \quad (2)$$

```
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++; }
}
```

Figure 2: Simplified RC4 implementation

**Verifying The Absence of Error Propagation in RC4.** We are now in a position to apply self-composition to verify whether the RC4 implementation in openSSL indeed satisfies the error propagation property. To do this, we use the approach introduced in Section 4. As has been pointed out in literature [17], this is a non-trivial exercise even for such a small example: the self-composition technique for the verification of non-interference is very attractive from a conceptual point of view, but its applicability to real-world applications is yet to be demonstrated. We believe the results in this section are an important contribution towards answering this open question. However, given that we have used an off-the-shelf, general-purpose, verification tool, which is still under development, problems were to be expected. The limitations that we encountered were essentially due to the growth of the problem size due to the aggressive optimisations that were used in the RC4 implementation in openSSL: (1) the use of macros rather than (inline) function calls leads to a source code size expansion; (2) the use of loop unrolling leads to intricate control flow inside the function (namely the extensive use of the break statement); and (3) the use of pointer arithmetic greatly increases the complexity of the generated proof obligations. For this reason, a refactoring of the code was required in order to achieve the goals that we set out in the beginning of this section (more on this in Section 7). Figure 2 shows the version of the RC4 we used.

The full version of this paper [1] contains the Frama-c input file used to show that the RC4 function above does not introduce error propagation. The function is composed with itself and disjoint sets of variables are created for the two copies of the function, which is parametrised with the position $i$ in which a transmission error could occur. The preconditions imposed on the composed function establish the equality of the high-integrity and unspecified-security input variables for both copies of the function: all input variables except position $i$ in the indata buffers. The post-condition on the composed function requires that the high-integrity output variables have equal values upon termination: $i+1$ to len in the outdata buffer. The verification of this code with Frama-c resulted in the generation of 17 proof obligations, all of which are automatically discharged by the prover Simplify. This is made possible by the inclusion of a helper lemma (proved offline with Coq following Section 4) in the ACSL annotations.

## 6 Related Work

Language Based Information Flow Security is surveyed in [15]. Leino and Joshi [12] were the first to propose a semantic approach to checking secure information flow, with several desirable features: it gives 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)[11] was presented by Warnier and Oostdijk[19]. They proposed an algorithm, based on the strongest post-condition calculus, that generates an annotated source file with specification patterns for confidentiality in JML. Dufay et al. [7] proposed an extension to JML to enforce non-interference through self-composition. This extended annotation language allows a simple definition of non-interference in Java programs. However, the generated proof obligations are complex, which limits the general applicability of the approach.

Terauchi and Aiken [17] identified problems in the self-composition approach, arguing that automatic tools are not powerful enough to verify this property over programs of realistic size. To compensate for this, the authors propose a program transformation technique based on an extended version of the self-composition approach, which incorporates the notion of security level downgrading using *relaxed non-interference* [13]. Rather than replicating the original code, the renamed version is interleaved and partially merged with it. Naumann[14] extended Terauchi and Aiken's work to encompass heap objects, presented a systematic method to validate the transformations proposed in [17], and reported on the experience of using these techniques with the ESC/JAVA2[10] and Spec# [2] tools.

# 7 Conclusions

The main contribution of this paper is to report on the application of the off-the-shelf `Frama-c` verification platform to 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 two security-relevant properties of this implementation: (1) safety properties such as the absence of numeric errors and memory safety, and (2) the absence of error propagation.

We take advantage of the built-in `Frama-c` functionality that automatically generates proof obligations for these two common types of secure coding safety properties. Concretely, we use `Frama-c` to prove that `RC4` implementation does not cause null pointer de-referencing exceptions, and always performs array accesses with valid indices. In other words, the implementation is secure against *buffer overflow* attacks. Additionally, we demonstrate 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 change may be due to a transmission error, or maliciously introduced by an attacker. 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 formalise this property as a novel application of the *non-interference* concept, widely used in the formalisation and verification of secure information flow properties. Subsequently, we proved that the `RC4` implementation indeed has this property.

Finally, our work answers some open questions raised by previous work, which seemed to indicate that self-composition was not directly applicable to real-world cases. Our results are promising in that we have been able to achieve our goal using an off-the-shelf verification tool and a technique with a high potential for automation. Our use of natural invariants is part of a bigger effort: the same technique facilitates the development of *equivalence proofs*, which we have been applying in the cryptographic context to prove the correctness of real implementations with respect to reference implementations. In particular, the validity of the refactoring used to produce the version of RC4 in Figure 2 can be addressed using these techniques. The full version of this work [1] covers also these aspects of our research.

## 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/`.

[2] 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.

[3] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE Computer Society, 2004.

[4] 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).

[5] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo: a theorem prover for polymorphic first-order logic modulo theories, 2006.

[6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[7] 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.

[8] 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.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[10] 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.

[11] 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.

[12] K. Rustan M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Lecture Notes in Computer Science*, 1422:254–271, 1998.

[13] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 158–170, New York, NY, USA, 2005. ACM.

[14] 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.

[15] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[16] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C.* Wiley, New York, 2nd edition, 1996.

[17] 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.

[18] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. `http://coq.inria.fr`.

[19] Martijn Warnier and Martijn Oostdijk. Non-interference in JML. Technical Report ICIS-R05034, Nijmegen Institute for Computing and Information Sciences, 2005.