Electronic Communications of the EASST
Volume 33 (2010)

Proceedings of the Fourth International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010)

A Deductive Verification Platform for Cryptographic Software

M. Barbosa, J. Pinto, J.-C. Filliâtre and B. Vieira

17 pages

Guest Editors: Luis S. Barbosa, Antonio Cerone, Siraj A. Shaikh
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Deductive Verification Platform for Cryptographic Software

M. Barbosa[1], J. Pinto[2], J.-C. Filliâtre[3] and B. Vieira[4]

[1]mbb@di.uminho.pt
[2]jsp@di.uminho.pt
[4]barbarasv@di.uminho.pt
CCTC/Departamento de Informática,
Universidade do Minho, Campus de Gualtar, Braga, Portugal

[3] Jean-Christophe.Filliatre@lri.fr
INRIA Saclay - Île-de-France, ProVal, Orsay, France
LRI, Université Paris-Sud, CNRS, Orsay, France

Abstract: In this paper we describe a deductive verification platform for the CAO language. CAO is a domain-specific language for cryptography. We show that this language presents interesting challenges for formal verification, not only in the rich mathematical type system that it introduces, but also in the cryptography-oriented language constructions that it offers. We describe how we tackle these problems, and also demonstrate that, by relying on the Jessie plug-in included in the Frama-C framework, the development time of such a complex verification tool could be greatly reduced. We base our presentation on real-world examples of CAO code, extracted from the open-source code of the NaCl cryptographic library, and illustrate how various cryptography-relevant security properties can be verified.

Keywords: formal program verification, cryptography

1 Introduction

Background. The development of cryptographic software is clearly distinct from other areas of software engineering. Cryptography is an inherently interdisciplinary subject. The design and implementation of cryptographic software draws on skills from mathematics, computer science and electrical engineering. However, there is a clear lack of domain specific languages and tools for the development of cryptographic software that can assist developers in facing these challenges. The CACE project (http://www.cace-project.eu) addresses this lack of support by pursuing the development of an open-source toolbox comprising languages, tools and libraries tailored to the implementation of cryptographic algorithms and protocols. The formal verification tool described in this work has been developed to allow the static analysis of code written in CAO, a new domain-specific language developed as a part of the CACE effort. Currently, this tool is being employed in the formal verification of an open-source library written in CAO. The results in this paper already reflect a part of this verification effort.

The CAO language[Bar09] allows practical description of cryptography-relevant programs. Unlike languages used in mathematical packages such as Magma or Maple, which allow the

description of high-level mathematical constructions in their full generality, CAO is restricted to enabling the implementation of cryptography kernels (e.g. block ciphers and hash functions) and sequences of finite field arithmetics (e.g. for elliptic curve cryptography). CAO has been designed to allow the programmer to work over a syntax that is similar to that of C, whilst focusing on the implementation aspects that are most critical for security and efficiency. The memory model of CAO is extremely simple (there is no dynamic memory allocation, there are no side-effects in expressions, and it has call-by-value semantics). Furthermore, the language does not support any input/output constructions, as it is targeted at implementing the core components in cryptographic libraries. On the other hand, the native types and operators in the language are highly expressive and tuned to the specific domain of cryptography. These languages feature can then be used by the CAO compiler to provide domain-specific analysis and optimization.

Deductive program verification and Frama-C. Program Verification is the area of Formal Methods that aims to statically check software properties based on the axiomatic semantics of programming languages. In this paper we focus on techniques based on Hoare logic, brought to practice through the use of *contracts* – specifications consisting of preconditions and postconditions, annotated into the programs. Verification tools based on contracts are becoming increasingly popular, as their scope evolved from toy languages to realistic fragments of languages like C, C#, or Java. We use the expression *deductive verification* to distinguish this approach from other ways of checking properties of programs, such as *model checking*.

In this work we build on Frama-C [BFM+08], an extensible framework where static analysis of C programs is provided by a series of plug-ins. Jessie [MM10] is a plug-in that can be used for deductive verification of C programs. Broadly speaking, Jessie performs the translation between an annotated C program and the input language for the Why tool. Why is a *Verification Condition Generator* (VCGen), which then produces a set of proof obligations that can be discharged using a multitude of proof tools that include the Coq proof assistant [The08], and the Simplify [DNS05], Alt-ergo [CCK06], and Z3 [MB08] automatic theorem provers. The gwhy graphical front-end, allows monitoring individual verification conditions. This is particularly useful when used in combination with the possibility of exporting the conditions to various proof tools, 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.

Motivation. Experience shows [ABPV09, ABPV10] that a tool such as Frama-C has a great potential for verifying a wide variety of security-relevant properties in cryptographic software implementations. However, it is well-known that the intrinsic characteristics of the C language make it a hard target for formal verification, particularly when the goal is to maximize automation. This problem is amplified when the verification target is in the domain of cryptography, because implementations typically explore language constructions that are little used in other application areas, including bit-wise operations, unorthodox control-flow (loop unrolling, single-iteration loops, break statements, etc.), intensive use of macros, etc. The idea behind the construction of a deductive verification tool for CAO is to take advantage of the characteristics of this programming language to construct a domain-specific verification tool, allowing for the same generic verification techniques that can be applied over C implementations, simplifying the verification of security-relevant properties, and hopefully providing a higher degree of automation.

Contributions. In this paper we describe an implementation of such a deductive verification platform for the CAO language. We show that CAO presents interesting challenges for formal verification, concerning not only the rich mathematical type system that it introduces, but also the cryptography-oriented language constructions that it offers. We describe how we tackle these problems, namely by presenting what we believe is the first formalisation in first-order logic of the rich mathematical data types that are used in cryptography in the context of deductive verification. We also demonstrate that, by relying on the Jessie plug-in of the Frama-C framework, the development time of such a complex verification tool could be greatly reduced. We base our presentation on real-world examples of CAO code, extracted from the open-source code of the NaCl cryptography library (http://nacl.cr.yp.to). The development of our tool has so far focused on automating safety verification, which we have achieved for a representative set of examples.

Organisation of the Paper. The next section expands on the application scenario and functional requirements for our tool. Section 3 describes the high-level implementation choices we have made. In Section 4 we introduce the most relevant parts of the translations performed by the tool. The generation of safety proof obligations is discussed separately in Section 5. Section 6 discusses related work, and Section 7 has some concluding remarks.

2 Deductive verification of CAO programs

A detailed specification of CAO can be found in [Bar09]. Appendix A includes an example we will use throughout the paper: a partial CAO implementation of the AES block cipher. As a C-like language, CAO supports analogous definitions of conditionals and loops, as well as global variable declarations, function declarations and procedures. The syntax of expressions is also similar to that of C, although the set of types and operations are significantly different.

The CAO type system includes a set of primitive types: arbitrary precision integers int, bit strings of finite length bits[n] , rings of residue classes modulo an integer mod[n] (intuitively, arithmetic modulo an integer, or a finite field of order n if the modulus is prime) and boolean values bool. Derived types allow the programmer to define more complex abstractions. These include the product construction struct, the generic one-dimensional container vector[n] of T, the algebraic notion of matrix, denoted matrix[i,j] of T, and the construction of an extension to a finite field T using a polynomial p(X), denoted mod[T<X>/p(X)]. Algebraic operators are overloaded so that expressions can include integer, ring/finite-field and matrix operations; the natural comparison operators, extended bit-wise operators, boolean operators and a well-defined set of type conversion (cast) operators are also supported. Bit string, vector and matrix access operations are extended with range selection (also known as slicing operations).

An implementation of a type-checker for CAO programs has been derived from the CAO type system formalisation [Bar09]. Hence, for the purpose of this paper we will assume that the CAO verification tool has access to an Abstract Syntax Tree (AST) with complete type information for the input CAO program. Note that this includes the concrete sizes of all container types, the moduli and polynomials in rings and finite fields, etc. Furthermore, the CAO type checker is able to reject all programs where incompatible type parameters are passed to an operator. For

example, the size restrictions associated with matrix addition and multiplication are enforced by the type system. The same happens for operations involving bit strings, rings and finite fields, where the type system checks that operator inputs have matching lengths, moduli, etc.

Safety in CAO. The verification that a program will not reach a point of execution that may result in a catastrophic failure, namely a run-time error, is commonly known as a *safety verification*. This type of verification goal is admittedly a modest one. Nevertheless, not only is safety verification a critical aspect for most practical applications, but also it is frequent that even this is a challenge for existing tools. In many cases, safety verification cannot be dealt with automatically, and it may become a labour-intensive activity. One of the requirements for the CAO verification tool is that safety verification should be feasible with minimum intervention from the end-user.

Program safety in CAO has two dimensions: memory safety and safety of arithmetic operations. A program is said to be memory safe if, at run-time, it never fails by accessing an invalid memory address. Memory safety verification is not in general a trivial problem in languages with pointers and heap-based data structures, and indeed there exist dedicated verification tools for this task. However, for CAO programs, this problem is reduced to making sure that all indices used in vector, bit string and matrix index accesses are within the proper range, which is fully determined by the type of the container and must be fully determined at compile time.

The safety of arithmetic operations is more interesting. In CAO we have four algebraic types: arbitrary precision integers, rings of residue classes modulo a non-prime, finite fields, and matrices thereof. The semantics of operators over these types is precisely given by the mathematical abstractions that they capture. This means that the concept of arithmetic overflow does not make sense in this context, and it leaves as candidate safety verification goals the possibility that such operators are not defined for some inputs. For integers, this reduces to the classic division-by-zero condition, whereas matrix addition and multiplication introduce are intrinsically safe.

Rings and finite fields pose an interesting problem, as they are not distinct CAO types. Take the following declarations:

```
def a : mod[13] := 4;          def b : mod[10] := 5;
def c : mod[13] := 1/a;        def d : mod[10] := 1/b;
```

All of these operations are safe, except for the initialization of d. The reason for this is that the multiplicative inverse modulo 10 is only defined for those integers in the range 1 to 9 that are co-prime with 10. This means that, whenever a division occurs in the mod[n] type, one must also ensure that the divisor is co-prime to the modulus.

When the modulus is a prime number, then the mod[n] type represents the finite field of size n. In this case, the previous problem reduces again to the division-by-zero case, as all non-zero elements have a multiplicative inverse. However, this observation does not help, unless there is a way to verify that the modulus is indeed a prime number. One way to do this, of course, is to allow the programmer to vouch for the primality of the modulus. We will return to this issue in Section 4. Finally, a related problem arises when one considers the construction of extension fields. In this case, not only must one ensure that the underlying type represents a finite field (which might not be the case for the mod[n] type) but also that the polynomial that is provided is irreducible in the corresponding ring of polynomials.

Extending CAO with annotations. CAO-SL is a specification language to be used in annotations added to CAO programs. These annotations are embedded in comments (so that they are ignored by the CAO compiler) using a special format that is recognised by the verification tool. CAO-SL is strongly inspired by ACSL [BFM+08] and enables the specification of behavioral properties of CAO programs. CAO-SL stands to CAO in the same way that ACSL stands to C.

The expressions used in annotations are called *logical expressions* and they correspond to CAO expressions with additional constructs. The semantics of the logical expressions is based on first-order logic. CAO-SL includes the definition of *function contracts* with pre- and postconditions, statement annotations such as assertions and loop variants and invariants, and other annotations commonly used in specification languages. CAO-SL also allows for the declaration of new logic types and functions, as well as predicates and lemmas. A complete description of CAO-SL can be found in [Bar09]. In this paper, various features of this language will be introduced gradually, as we describe the tool architecture and implementation.

3 Tool implementation

Our tool follows the same approach used in other scenarios for general-purpose languages such as Java [MPU04] and C [FM04]. Furthermore, the tool architecture itself fundamentally relies on the Jessie plug-in included in the Frama-C framework. This allowed us to significantly reduce the tool development time and effort. Jessie enables reasoning about typical imperative programs, and it is equipped with a first-order logic mechanism, which facilitates the design of new models and extensions. In particular, it is possible to use this feature to define in Jessie a model of the domain-specific types and memory model of CAO. This means that an annotated CAO program can be translated into an annotated Jessie program and, from this point on, our verification tool can rely totally on the functionality of Jessie and Why. The translation is such that correctness of the Jessie program entails the correctness of the source CAO program.

The Jessie Input Language. The Jessie input language is a simply typed imperative language with a precisely defined semantics. It is used as an intermediate language for verification of C programs in the Frama-C framework. As an intermediate language, programmers are not expected to produce Jessie source programs from scratch. Jessie was developed in parallel with ACSL, and they share many constructions. The language combines operational and logic features. The operational part refers to statements which describe the control flow and instructions that perform operations on data, including memory updates. The logical part is described through formulas of first-order logic, attached to statements and functions in the form of annotations. Jessie provides primitive types such as integers, booleans, reals and unit (the void type), abstract datatypes and also allows the definition of new datatypes.

Programs can be annotated using pre- and postconditions, loop invariants, and other intermediate assertions. The logical language is typed and includes built-in equality, booleans, arbitrary precision integers, and real numbers.

Implementation strategy. In order to better illustrate our approach to designing a VCGen for CAO taking advantage of an existing *generic* VCGen, we introduce a very simple example.

Consider the definition of a VCGen for the subset of CAO that is essentially a *while* language with applicative arrays[1], and how one would deal with both *aliasing* and *safety*. The weakest precondition of the array assignment operation would resemble the following

$$\mathsf{wp}(u[e] := x, Q) \;=\; \mathit{safe}(u[e]) \wedge \mathit{unalias}(Q, e, x)$$

where $\mathit{safe}(u[e]) \;=\; \mathit{safe}(e) \wedge 0 \leq e < \mathit{length}(u)$, $\mathit{safe}(e)$ imposes that the evaluation of e will not produce arithmetic errors, and the function *unalias* would process Q, updating the contents of index e to x considering aliasing, e.g.

$$\mathit{unalias}(u[j] > 100, i, 10) \;=\; \quad (i = j \implies 10 > 100) \wedge (i \neq j \implies u[j] > 100)$$

If one momentarily forgets safety considerations, an alternative possibility is to construct the VCGen on top of an existing VCGen for the base *while* language, not by adding new dedicated rules, but instead by translating the new type being introduced as a logical type. Applicative arrays can be modeled by the following well-known axioms for the *get* and *set* logical functions.

$$\forall u, e, x.\ \mathit{get}(\mathit{set}(u, e, x), e) = x \tag{1}$$

$$\forall u, e, e', x.\ e \neq e' \implies \mathit{get}(\mathit{set}(u, e, x), e') = \mathit{get}(u, e') \tag{2}$$

In the presence of this theory, one can use the VCGen for the core *while* language to derive verification conditions (VCs) for array lookup expressions and assignment commands, by simply translating these to use the definitions above:

$$\mathscr{T}(u[e]) \;\equiv\; \mathit{get}(u, \mathscr{T}(e)) \qquad\qquad \mathscr{T}(u[e] := x) \;\equiv\; u := \mathit{set}(u, \mathscr{T}(e), \mathscr{T}(x))$$

A powerful generic VCGen such as Jessie allows us to follow this approach for the entire CAO language. There is an overlap between the CAO language and the Jessie input language that enables a direct translation of many language constructions. Furthermore, for each CAO type that is not supported by Jessie, we are able to declare a set of logical functions and write a theory for them that creates a suitable first-order model of the type. This then enables us to translate arbitrary annotated CAO programs into suitable programs of the Jessie input language.

Let us now turn back to the example above, to see how we deal with safety conditions using Jessie's *assert* clause to force the generation of arbitrary proof obligations. Safety conditions for applicative arrays can be generated by using the following translation:

$$\begin{aligned} \mathscr{T}(u[e]) \;&\equiv\; \textbf{assert}\ 0 \leq \mathscr{T}(e) < \mathit{length}(u);\ \mathit{get}(u, \mathscr{T}(e)) \\ \mathscr{T}(u[e] := x) \;&\equiv\; \textbf{assert}\ 0 \leq \mathscr{T}(e) < \mathit{length}(u);\ u := \mathit{set}(u, \mathscr{T}(e), \mathscr{T}(x)) \end{aligned}$$

Of course, in CAO we have to deal with data types that are considerably more sophisticated than arrays. Yet, the general pattern followed in the implementation of our tool is the same. The introduction of each new type implies the introduction of a new theory. The definition of a new theory includes the definition of logic functions together with axioms to model their behavior. Some lemmas and predicates are also introduced to speed up the process of proving some goals.

At this point, it makes sense to ask which properties of those types should be included in the corresponding logical model. *Soundness* is of course the most important property that should be guaranteed by our translation process: the Jessie model should not allow proving assertions about CAO data structures that are not valid according to the language semantics. A second very

[1] We use this denomination for typical imperative arrays with destructive update, and with opaque storage.

desirable property is that the model should allow for as many assertions as possible to be proved automatically. More precisely, the verification conditions produced by Jessie, and exported to some external theorem prover, should as much as possible be discharged automatically.

Emphasis on Automation. The fact that Jessie relies on the Why VCGen, which is a *multi-prover* tool, means that it is possible to export verification conditions to a large number of different proof tools, from SMT-solvers to the Coq interactive proof assistant. The typical workflow is to first discharge "easy" VCs using an automatic prover, and then interactively trying to discharge the remaining conditions. Once the model of CAO is fixed, different properties of CAO code will naturally have different degrees of automation with respect to discharging VCs. As is true of VCGens for other realistic languages, safety conditions should be proved with a high degree of automation, whereas a lower degree should be expected for other functional properties. Our approach is multi-tiered in the sense that we start with high-level models tuned for automatic verification (in particular of safety properties); these models can then be refined into lower-level models that take advantage of theories supported by specific automatic provers (such as bit strings, integers, and so on). Finally, all models can be further refined to Coq models: interactive proof is the last resort for discharging VCs, and it may be mandatory for any verification tool based on first-order logic.

Our efforts have so far concentrated on maximizing the degree of automation that we can achieve in verifying the safety of CAO programs. We are able, for example, to carry out the safety verification of the entire CAO implementation of the AES block-cipher (see Appendix) without user intervention. This includes heavy use of finite field, vector and matrix operations across several dependent functions.

4 CAO to Jessie translation

We will resort to snippets of CAO code to describe the most interesting parts of the CAO to Jessie translation carried out by our verification tool, which essentially correspond to the rich cryptography-specific data types that are available in CAO. In other words, we will focus on the way in which we handle the parts of the CAO language (including the extension to CAO-SL) that do not directly map to constructions in the Jessie input language, leaving out the standard imperative constructions supported by both languages, the CAO types that directly map to Jessie native types, and the translation of annotations. In the following $\lceil e \rceil$ denotes the translation of a CAO expression e into Jessie.

4.1 Container Types

The container types in CAO include the vector[] of, matrix[] of and bits types. The get and set operations on these types are modeled in Jessie using exactly the second approach that we described in the example in the previous section. The only caveat is that they are generalized to the two dimensional case in the case of matrices, and that we fix Jessie type bool as the content type in the case of bits.

However, CAO includes elaborate operators to deal with these container types that are fine-

$$
\begin{array}{l}
blit\_vector\_\lceil\tau\rceil \ : \ vector\_\lceil\tau\rceil \rightarrow \ vector\_\lceil\tau\rceil \rightarrow integer \rightarrow integer \rightarrow \ vector\_\lceil\tau\rceil \\
shift\_vector\_\lceil\tau\rceil \ : \ vector\_\lceil\tau\rceil \rightarrow integer \rightarrow vector\_\lceil\tau\rceil \\
\\
\forall v, ofs, i.\ get\_vector\_\lceil\tau\rceil(shift\_vector\_\lceil\tau\rceil(v, ofs), i) = get\_vector\_\lceil\tau\rceil(v, (ofs + i)) \\
\forall src, dest, ofs, len, i.\ ofs \leq i < (ofs + len) \implies \\
\qquad get\_vector\_\lceil\tau\rceil(blit\_vector\_\lceil\tau\rceil(src, dst, ofs, len), i) = get\_vector\_\lceil\tau\rceil(src, i - ofs) \\
\forall src, dest, ofs, len, i.\ i < ofs \ \vee \ i \geq (ofs + len) \implies \\
\qquad get\_vector\_\lceil\tau\rceil(blit\_vector\_\lceil\tau\rceil(src, dst, ofs, len), i) = get\_vector\_\lceil\tau\rceil(dst, i)
\end{array}
$$

Figure 1: Declarations and axioms for vector types.

tuned to the implementation of cryptographic algorithms, namely symmetric primitives such as block ciphers and hash functions. As an example, consider the next snippet from the AES implementation in CAO.

```
def ShiftRows( s : S ) : S {
  def r : S;
  seq i := 0 to 3 { r[i,0..3] := (Row)(((RowV)s[i,0..3]) |> i); }
  return r; }
```

What we have here is a sequence of rotation (|>) operations applied to the i^{th} row of matrix s. The way in which this is expressed in CAO takes advantage of the range selection operator (..) that returns a value of the corresponding container type, with the same contents as the original one, but with appropriate dimensions. Here, this operator is used to select an entire row in the matrix, which is cast to the vector type in order to be rotated. The result is then cast back to the correct matrix type that can be assigned to the original row slice in matrix r.

Our first-order formalisation of container types deals with shift, rotate, range selection, range assignment and concatenation (@) operators in container types using a pattern that relies on two logic functions (shift and blit). We present the case of the vector type. The model assumes that a vector has infinite length, i.e., it has a start position, but it is represented as an unbounded (infinite length) memory block. The only exception to this rule is the extensional equality operator (==), where translation explicitly refers to the range of valid positions over which equality should hold. We emphasize that this part of the model deals only with the functionality of these operators: safety is handled separately by introducing appropriate assertions, as will be seen in Section 5.

Intuitively, the *shift* logic function takes as input a vector of arbitrary length, starting in position 0, and produces the sub-vector that starts at position i. The *blit* logic function involves two vectors, source s and destination d, an index i and a length parameter l. It produces the vector with the contents of d for indices 0 to i-1, and from i+l onwards; the l positions in between contain the region 0..l-1 of s. The behaviour of these logic functions is modeled by the declarations and axioms given in Figure 1.

Range Selection. Given a CAO variable μ of type $vector[n]\ of\ \tau$, the CAO range selection operation is modeled in Jessie as follows:

$$
\begin{array}{l}
\lceil\ \mu[i..j]\ \rceil \ \equiv \textbf{let}\ x_1 = \lceil i \rceil\ \textbf{in}\ (\ \textbf{let}\ x_2 = \lceil j \rceil\ \textbf{in} \\
\qquad\qquad \textbf{assert}\ (0 \leq x_1 < n)\ \&\&\ (0 \leq x_2 < n)\ \&\&\ (x_1 \leq x_2);\ shift(\lceil \mu \rceil, x_1))
\end{array}
$$

where i and j are integer expressions. We remark that although the translation disregards the upper bound j, the typechecker will ensure that the range selection operation $\mu[i..j]$ with μ of type $vector[n]\ of\ \tau$, returns type $vector[j-i+1]\ of\ \tau$, thus taking that upper bound into account.

Range assignment. Assigning to a region in a vector is modeled directly using the blit function.

$$\begin{aligned}\lceil \mu_1[i..j] := \mu_2 \rceil \equiv\ & \textbf{let}\ x_1 = \lceil i \rceil\ \textbf{in}\ (\ \textbf{let}\ x_2 = \lceil j \rceil\ \textbf{in} \\ & \textbf{assert}\ (0 \le x_1 < n)\ \&\&\ (0 \le x_2 < n)\ \&\&\ (x_1 \le x_2); \\ & \lceil \mu_1 \rceil\ =\ blit(\lceil \mu_2 \rceil, \lceil \mu_1 \rceil, x_1, x_2 - x_1 + 1))\end{aligned}$$

Here, $=$ denotes assignment in Jessie.

Concatenation. Consider the CAO variables μ_1 and μ_2 of types $vector[n_1]\ of\ \tau$ and $vector[n_2]\ of\ \tau$ respectively. The concatenation of vectors μ_1 and μ_2 can also be captured using the blit function.

$$\lceil \mu_1\ @\ \mu_2\ \rceil \equiv blit(\lceil \mu_2 \rceil, \lceil \mu_1 \rceil, n_1, n_2)$$

The intuition behind this definition is that concatenation can be seen as a range assignment operation, where μ_2 is assigned to the region of μ_1 that starts at position n_1 (recall that in the model vectors are assumed to have infinite length).

Shift and Rotate. To present the shift and rotate operations in a more intuitive way, we will turn to the bits type. Both operations are modeled using the blit function.

The rotate operations are commonly known as circular shifts. A downwards circular shift by 1 is defined as a permutation of the entries in a tuple where the last element becomes the first element and all the other elements are down-shifted one position. Conversely, in an upwards circular shift, the first element becomes the last element and all the other are shifted up. As an example, consider the bits literal: `0b1101001`. The internal representation of bits in our model stores the least significant bit (the right-most bit in the literal) in the 0-th position. This means that upwards and downwards rotate correspond to the intuitive interpretation of left and right rotations, respectively. An example of a down rotate is therefore $0b1101001 \mid > 3 = 0b\underline{001}1101$ and an example of an up rotate is $0b1101001 < \mid 3 = 0b1001\underline{110}$. In our model, for a CAO expression e of type $vector[n]\ of\ \tau$ or $bits[n]$, we have:

$$\begin{aligned}\lceil e <\mid i \rceil &\equiv \lceil e[n-i\ ..\ n-1]\ @\ e[0\ ..\ n-i-1] \rceil \equiv blit(shift(\lceil e \rceil, 0), shift(\lceil e \rceil, n-i), i, n-i) \\ \lceil e \mid > i \rceil &\equiv \lceil e[i\ ..\ n-1]\ @\ e[0\ ..\ i-1] \rceil \equiv blit(shift(\lceil e \rceil, 0), shift(\lceil e \rceil, i), n-i, i)\end{aligned}$$

where i is a constant of type int. The intuition is that rotations can be seen as concatenations of the appropriate sub-regions, which in turn are modeled using the *blit* function.

Logical shifts are handled in a similar way, but resorting to $bits\_null\_vector$ (the all-zeroes bits value) to fill in the positions left vacant by the operation.

Matrices. Our model of matrices for the equivalent vector operators described above is essentially a direct generalization of the above strategy to the 2-dimensional case. However, our model of matrices must also account for the fact that the matrix type in CAO is an algebraic type that

supports addition and multiplication operations (indeed this is why in CAO you can only define matrices whose contents are themselves algebraic types).

The formalisation of matrices in first-order logic includes the matrix addition and multiplication arithmetic operations as logic functions

$$matrix\_\lceil\tau\rceil\_add,\ matrix\_\lceil\tau\rceil\_mult\ :\ matrix\_\lceil\tau\rceil \to matrix\_\lceil\tau\rceil \to matrix\_\lceil\tau\rceil$$

The functionality of the addition operator is modeled using the following axiom:

Axiom 1 *Let A and B be matrices of dimensions $m \times n$, and a_{ij} and b_{ij} the elements in the i^{th} row and j^{th} column of A and B, respectively. Then, $\forall\ j,i.\ (A+B)_{ij} = a_{ij} + b_{ij}$.*

An equivalent axiom for matrix multiplication was not introduced because, for each possible base type, we would need the (higher-order) logic formalization of summation Σ.

The translation of expressions with arithmetic operations of type $matrix[n_1,n_2]\ of\ \tau$ is therefore the following:

$$\lceil\ \mu_1\ +\ \mu_2\ \rceil = matrix\_\lceil\tau\rceil\_add(\lceil\mu_1\rceil,\lceil\mu_2\rceil) \qquad \lceil\ \mu_1\ *\ \mu_2\ \rceil = matrix\_\lceil\tau\rceil\_mult(\lceil\mu_1\rceil,\lceil\mu_2\rceil).$$

Bitwise operations. We complete this section with a brief description of how bitwise operations are handled in our model, as these are of critical importance in cryptographic applications. Here we greatly benefit from the design of the CAO language, where the classic ambivalence between integers and their bit-level representations that exists in the C int type is eliminated by introducing the bits type. Indeed, CAO programmers can freely use bit strings of any size, and convert these to and from the type int that represents the mathematical type $\mathbb{Z}$. A very simple model of bit strings based on vectors of bits (boolean values) can be used, although things get more complicated when we need to deal with type conversions. The Jessie model of bitwise operations on bits is based on the following logic functions:

$$bits\_bitwise\_xor : bits \to bits \to bits \qquad bits\_bitwise\_and : bits \to bits \to bits$$
$$bits\_bitwise\_or : bits \to bits \to bits \qquad bits\_bitwise\_neg : bits \to bits$$

which are axiomatized in the obvious way. CAO bitwise operations are translated as:

$$\lceil e_1\ \oplus\ e_2\rceil \equiv bits\_bitwise\_\lceil\oplus\rceil(\lceil e_1\rceil,\lceil e_2\rceil) \qquad \lceil !\ e\rceil \equiv bits\_bitwise\_neg(\lceil e\rceil)$$

where $\oplus \in \{|,\&,\hat{}\}$ and μ_1 and μ_2 are expressions of type $bits[n]$.

4.2 Rings, fields and extension fields

Residue classes modulo n. The mod[n] type is an algebraic type. For $n \in \mathbb{N}$, it corresponds to the algebraic ring $\mathbb{Z}_n$. Moreover if n is prime, then mod[n] permits programmers to take full advantage of the fact that $\mathbb{Z}_n$ is a field. The Jessie model for the mod[n] type is based on the theory of integers, taking advantage of optimized models supported by many automatic provers, and fully integrated into Jessie.

The model of mod[n] starts with the definition of the logic type mod_n equipped with logic functions corresponding to the two natural homomorphisms that convert to and from the Jessie integer type, as well as the mapping that results from their composition.

$int\_of\_mod_n : mod_n \rightarrow integer$ $\quad$ $mod_n\_of\_int : integer \rightarrow mod_n$ $\quad$ $mod_n : integer \rightarrow integer$

Hence, mod_n represents the mapping from $\mathbb{Z}$ to $\mathbb{Z}$ that associates to each $a \in \mathbb{Z}$ the least residue $r \in \mathbb{Z}$ of $[a]$. The model is then extended with a set of axioms for the following mathematical properties of these functions

$$\forall x.\ 0 \leq int\_of\_mod_n(x) \leq n-1 \qquad \forall x.\ 0 \leq x \leq n-1 \implies mod_n(x) = x$$
$$\forall x.\ mod_n(int\_of\_mod_n(mod_n\_of\_int(x)) = mod_n(x) \qquad \forall x.\ x \geq n \implies mod_n(x) = mod_n(x-n)$$
$$\forall x.\ x < 0 \implies mod_n(x) == mod_n(x+n)$$

The Jessie translation of arithmetic operations involving expressions of type mod[n] is based on the homomorphisms declared above. First, $int\_of\_mod_n$ is used to get the least residues of the equivalence classes involved in the arithmetic operation. These least residues are integers, which allows us to model the arithmetic operations using the integers theory. Finally, we apply $mod_n\_of\_int$ to the result to recover the equivalence class that represents that value. Hence, the translation of arithmetic operations on type mod[n] is given as follows for $op \in \{+, -, *, **\}$.

$$\lceil e_1\ op\ e_2 \rceil \equiv mod_n\_of\_int(int\_of\_mod_n(\lceil e_1 \rceil)\ \ op_{int}\ \ int\_of\_mod_n(\lceil e_2 \rceil))$$
$$\lceil e_1\ /\ e_2 \rceil \equiv \textbf{let}\ x = int\_of\_mod_n(\lceil e_2 \rceil)\ \textbf{in}$$
$$\textbf{assert}\ gcd(x,n) = 1;\ mod_n\_of\_int(int\_of\_mod_n(\lceil e_1 \rceil) *_{int} inv\_mod(x,n))$$

Note the special case of division. This is justified because the semantics of division modulo n is not the same as integer division. Firstly, one must express the correct semantics, which we do by introducing the logical function $inv\_mod(x,n)$. Simple properties involving operations with this function, which are used to easily discharge some proof obligations, are axiomatized as follows:

$$\forall x.\ gcd(int\_of\_mod_n(x)),n) = 1 \implies mod_n(int\_of\_mod_n(x) *_{int} inv\_mod(int\_of\_mod_n(x),n)) = mod_n(1)$$

$$\forall x,y.\ mod_n(int\_of\_mod_n(x) *_{int} y) = mod_n(1) \implies inv\_mod(int\_of\_mod_n(x),n) = mod_n(y)$$

Secondly, in the division case, one must generate a proof obligation for the safety condition that CAO programs should not perform undefined divisions. This property is trivially true if the divisor is in the range $1 \ldots n$-1 and the number n is prime. Hence we add the following axiom to our model, to automatically handle these trivial cases.

$$\forall x,n.\quad is\_prime(n)\ \wedge\ (0 < x < n) \implies gcd(x,n) = 1$$

where $is\_prime : integer \rightarrow boolean$ is a predicate to check if an integer number is prime, and $gcd : integer \rightarrow integer \rightarrow integer$ is a logic function that calculates the greatest common divisor between two integer numbers. Note that *is_prime* and *gcd* are neither directly defined nor axiomatized, but the programmer can explicitly assert that some n is prime through a CAO-SL annotation. This enables automatically discharging safety assertions using *gcd*.

Extension Fields. Consider the following type declarations taken from the AES implementation:

```
typedef GF2  := mod[2];
typedef GF2N := mod[GF2<X> / X**8+X**4+X**3+X+1];
typedef GF2C := mod[GF2N<Y> / Y**4+1];
```

Take the first field extension type GF2N. Types of the form mod[mod[n] <X>/p(X)] are also algebraic types that model the Galois field of order n^d where n is a prime number and d is the degree of the irreducible polynomial $p(X)$. We emphasize that in CAO each such type represents a specific construction of an extension field, whose representation is fixed as elements of the polynomial ring $\mathbb{Z}_n[X]$, and the semantics of operations is defined based on polynomial arithmetics modulo $p(X)$. Furthermore this type is only valid when n is prime and $p(X)$ is irreducible.

The theory of extension fields of this form begins with the definition of a logic type $ring\_mod_n$ that represents the ring of polynomials over the base type mod[n] and logic functions to model the elements of the ring and the addition operation that permits combining them.

$$ring\_mod_n\_monomial : mod_n \rightarrow integer \rightarrow ring\_mod_n$$
$$ring\_mod_n\_add : ring\_mod_n \rightarrow ring\_mod_n \rightarrow ring\_mod_n$$

Our model explicitly captures the fact that elements of this ring are polynomials, which in turn can be defined as an addition of monomials. The reason for this is that the CAO literal that corresponds to the irreducible polynomial $field\_mod_n\_poly_{f(x)}\_generator$ used to construct these types can then be represented in our logical model. A monomial can be represented by its coefficient (which is an element of mod_n) and its degree (an integer). Arithmetic operations over the polynomial ring are not included in the model, as they do not exist in CAO. Indeed our model is purposefully incomplete because we do not intend to use automatic theorem provers on verification conditions involving arbitrary extension field algebra. The goal is to use specific interactive proof assistants, namely Coq, to prove these kinds of properties, relying on existing libraries (e.g. SSReflect[2]) that provide theories for abstract algebra (fields, polynomials, etc).

The model is then completed by adding definitions for the type $field\_mod_n\_poly_{f(x)}$ and the corresponding arithmetic operations. The Jessie translation of the arithmetic operations defined over mod[mod[n] <X>/p(X)] is then a direct one:

$$\lceil e_1\ op\ e_2 \rceil \equiv \lceil e_1 \rceil\ op_{field\_mod_n\_poly_{f(x)}}\ \lceil e_2 \rceil$$
$$\lceil e_1\ /\ e_2 \rceil \equiv \textbf{let}\ x = \lceil e_2 \rceil\ \textbf{in assert}\ x \neq 0_{field\_mod_n\_poly_{f(X)}};\ \lceil e_1 \rceil\ div_{field\_mod_n\_poly_{f(X)}}\ x$$

where $op \in \{+, -, *, **\}$. Note that there is also a special case for division. This ensures that a safety proof obligation is generated that checks if the divisor is different from zero.

A set of axioms that describe basic properties of these operators has been added to the model in order to increase the degree of automation provided by our tool. The goal here is that, given that there is no integrated support for this sort of mathematical construction in the automatic provers interfaced with Jessie, some simple properties can be captured in first order logic that permit dealing with trivial deduction steps, e.g. cancellation rules. The following axioms are included in our model

$$\forall a, b.\ a \neq 0_F \wedge b \neq 0_F \implies a \times_F b \neq 0_F \qquad \forall a, b.\ a \neq 0_F \implies a\ div_F\ b \neq 0_F$$
$$\forall a, b.\ a \neq b \implies a -_F b \neq 0_F \qquad \forall a, b.\ a \neq -b \implies a +_F b \neq 0_F$$
$$\forall a, b.\ a \neq 0_F \implies a\ (**)_F\ b \neq 0_F \qquad \forall a.\ a \neq 0_F \implies -_F a \neq 0_F$$

where $F = field\_mod_n\_poly_{f(X)}$. Literals of the extension field types are modeled in Jessie as

[2] http://www.msr-inria.inria.fr/Projects/math-components

$$
\begin{array}{rcl}
bits[n] & \Rightarrow & int \\
mod[n] & \rightarrow & int \\
int & \rightarrow & bits[n] \\
\tau & \Rightarrow & mod[\tau < X > /f(X)] \\
vector[n]\ of\ \tau & \rightarrow & mod[\tau < X > /f(X)]
\end{array}
\qquad
\begin{array}{rcl}
mod[\tau < X > /f(X)] & \rightarrow & vector[n]\ of\ \tau \\
matrix[1,n]\ of\ \tau & \rightarrow & vector[n]\ of\ \tau \\
matrix[n,1]\ of\ \tau & \rightarrow & vector[n]\ of\ \tau \\
vector[n]\ of\ \tau & \rightarrow & matrix[1,n]\ of\ \tau \\
vector[n]\ of\ \tau & \rightarrow & matrix[n,1]\ of\ \tau
\end{array}
$$

Figure 2: Casts and coercions

vectors of polynomial coefficients. Therefore, logic functions to access and update the coefficient of a given power of some polynomial of type mod[mod[n] <X>/p(X)] are also included in the model, together with the usual two axioms for the theory of arrays.

$$field\_mod_n\_poly_{f(x)}\_get\_coef : field\_mod_n\_poly_{f(x)} \rightarrow integer \rightarrow mod_n$$

$$field\_mod_n\_poly_{f(x)}\_set\_coef : field\_mod_n\_poly_{f(x)} \rightarrow integer \rightarrow mod_n \rightarrow field\_mod_n\_poly_{f(x)}$$

Returning to the example introduced above, it can be seen by examining the type declaration of GF2C that the base type of an extension field can actually be an extension field itself. However, our modeling approach is exactly the same for this case, taking into consideration that the base type must be adjusted when defining the ring of polynomials over the base field.

4.3 Casts and Coercions

Type conversion operations in CAO can be explicit, in which case they are called *cast* operations, or implicit, called *coercion* operations. Figure 2 presents the allowed cast ($\rightarrow$) and coercion ($\Rightarrow$) operations between CAO types. The translation of CAO programs into Jessie handles these conversions in the natural way by using appropriate logical functions. We present a few examples of the simpler conversions:

$$
\begin{array}{l}
e :: mod[n] \implies \lceil (int)\ e \rceil = int\_of\_mod_n(\lceil e \rceil) \\
e :: int \implies \lceil (mod[n])\ e \rceil = mod_n\_of\_int(\lceil e \rceil) \\
e :: int \implies \lceil (bits[n])\ e \rceil = bits\_of\_int(\lceil e \rceil) \\
e :: \tau \implies \lceil (mod[\tau{<}X{>}/f(X)])\ e \rceil = field\_\lceil \tau \rceil\_poly_{f(x)}\_set\_coef(field\_\lceil \tau \rceil\_poly_{f(x)}\_zero, 0, \lceil e \rceil_\tau)
\end{array}
$$

Conversions between matrices and column/row vectors are handled in the natural way by using get and set operations. Finally, we present the conversion between extension field types and vector types in a bit more detail, since these are very useful CAO operators that permit commuting between the abstract algebraic view of a finite field, and its concrete representation in a cryptographic implementation. Indeed, one can construct an extension field value from a vector representation that contains the coefficients of the corresponding polynomial over the base field. We model this as

$$
\begin{array}{l}
\lceil (mod[\tau < X > /f(X)])\ e \rceil = \\
\quad \textbf{let}\ x_1 = field\_\lceil \tau \rceil\_poly_{f(x)}\_zero\ \textbf{in}\ (\ \textbf{let}\ x_2 = \lceil e \rceil\ \textbf{in} \\
\quad \textbf{let}\ x_3 = field\_\lceil \tau \rceil\_poly_{f(x)}\_set\_coef(x_2, n-1, vector\_\lceil \tau \rceil\_get(x_2, n-1))\ \textbf{in}\ ... \\
\quad \textbf{let}\ x_{n+2} = field\_\lceil \tau \rceil\_poly_{f(x)}\_set\_coef(x_{n+1}, 0, vector\_\lceil \tau \rceil\_get(x_2, 0))\ \textbf{in}\ x_{n+2})
\end{array}
$$

The inverse conversion is also possible, and is modeled using a similar approach. This translation further justifies our modeling of extension field literals presented in the previous section.

Table 1: Safety proof obligations

Type	Operation	Proof Obligation	Auto
$integer$	p_1/p_2	$p_2 \neq 0$	×
$mod[n]$	p_1/p_2	$gcd(int\_of\_mod_n(p_2),n) = 1;\ int\_of\_mod_n(p_2) \neq 0$	×
$mod[n] < X > /f(X)$	$p_1\ /\ p_2$	$p2 \neq 0$	
$vector[n]\ of\ \tau$	$v[e]$ $v \rhd i, v \lhd i$ $v[i..j]$	$0 \leq \lceil e \rceil < n$ $0 \leq \lceil i \rceil < n$ $0 \leq \lceil i \rceil < n \wedge 0 \leq \lceil j \rceil < n \wedge \lceil i \rceil < \lceil j \rceil$	
$matrix[n_1,n_2]\ of\ \tau$	$m[e_1,e_2]$ $m[i..j,k..l]$	$0 \leq \lceil e_1 \rceil < n_1\ \wedge\ 0 \leq \lceil e_2 \rceil < n_2$ $0 \leq \lceil i \rceil < n_1\ \wedge\ 0 \leq \lceil j \rceil < n_1\ \wedge 0 \leq \lceil k \rceil < n_2\ \wedge$ $0 \leq \lceil l \rceil < n_2\ \wedge\ \lceil i \rceil < \lceil j \rceil\ \wedge\ \lceil k \rceil < \lceil l \rceil$	
$bits[n]$	$b[e]$ $b \rhd i, b \lhd i, b \gg i, b \ll i$	$0 \leq \lceil e \rceil < n$ $0 \leq \lceil i \rceil < n$	

5 Automatic safety proof obligations

Following the same approach adopted in tools such as Frama-C, the CAO to Jessie translation in our tool ensures that all statements in the input program that could potentially result in a safety violation originate the automatic generation of a verification condition that, if proven, guarantees the safe execution of the verified code.

We have two classes of safety proof obligations: those related with memory safety, and those related with algebraic type declarations and operations. Some of the proof obligations are automatically generated by the Jessie tool, while others are explicitly introduced in the generated Jessie code as assertions, during the translation process. We have encountered examples of these assertions in the models for division operations presented above. Table 1 presents the proof obligations that are generated to ensure the safety of memory access and algebraic operations. Proof obligations automatically generated by the Jessie plug-in are signaled in the table, corresponding to those that originate from the use of the Jessie integer type.

The safety proof obligations that are generated when types are declared are limited to the declaration of extension fields of the form mod[mod[n] <X>/p(X)]. In this case, the proof depends on the two following generated lemmas.

$$is\_prime(n) \qquad ring\_mod_n\_is\_irreducible(field\_mod_n\_poly_{f(x)}\_generator)$$

These are required to allow the automatic discharge of some proof obligations, but they also ensure that the user is aware of the type declarations and their implications. Lemmas can be immediately used in proofs, so for instance the first lemma can be used as an hypothesis in all proof obligations related to division operations in mod[n], requiring that the divisor is relative prime to the modulus. We emphasise however that the presence of lemmas also originates new proof obligations corresponding to the validation of the lemmas themselves.

6 Related work

The verification infrastructure introduced in the Jessie plug-in was already used in the development of other verification tools for C and Java. Caduceus [FM04], a tool for C, and Krakatoa [MPU04], a tool for Java, are also built on the top of Why tool. The translation into Why performed by Krakatoa is similar to that performed by Frama-C and also adopted in this paper.

Boogie [BED+06] is a verification condition generator very similar to Why. The input languages to Boogie and Why are both languages with imperative features and first-order propositions. In both cases, verification condition generation is based on the weakest precondition calculus. Boogie has front-ends for extensions of C# and C which enrich the languages with annotations in first-order logic, such as pre- and postconditions, assertions and loop invariants. The C# extension is known as Spec# [BMF+]. Boogie performs loop-invariant inference using abstract interpretation and then generates the verification conditions for Simplify or Z3. The core component of VCC [CDH+09], a tool for low-level concurrent C programs, is also Boogie.

Esc/Java [FLL+02] is another deductive verification tool for Java programs whose annotation language is a subset of JML [LRL+00]. Its architecture is similar to the ones presented above and based on an earlier checker for the Modula-3 language. This tool relies on loop unrolling, and the fact that generation of verification conditions includes optimizations to avoid the exponential blow-up inherent in a naive weakest-precondition computation. It looks for run-time errors in annotated Java programs, but does not model arithmetic overflow.

Jack (Java Applet Correctness Kit) [BBC+07] is a static verification tool for JML-annotated programs. It provides support for annotation generation and for interactive verification of functional specifications, as well as support for automatic verification of common security policies and for verification of byte-code programs. Its integration in the Eclipse IDE allows for proof obligation inspection, allowing users to visualize where in the code they are originated.

7 Conclusions

We have presented a model in first-order logic of certain mathematical objects, taking advantage of the theories implemented in general Satisfiability Modulo Theories (SMT) solvers. The objects that we model have specific interest for cryptographic security and for the verification of CAO programs in particular, but we believe that the model has independent interest and can be of use in other areas, whenever formal verification involving these objects is important.

Admittedly, the fact that the tool has not been proved correct is a flaw. Note however that this work is part of a bigger effort on the formalization of the CAO programming language. In related work we are working on an operational semantics for CAO, which we will later use to establish a correctness result for our VCGen. We remark however that the reliability of the VCGen is already high, since we rely on Jessie to capture the semantics of the basic language aspects of CAO, such as control structures and mutually recursive, contract-annotated procedures.

References

[ABPV09] J. B. Almeida, M. Barbosa, J. S. Pinto, B. Vieira. Verifying Cryptographic Software Correctness with Respect to Reference Implementations. In *Formal Methods for Industrial Critical Systems (FMICS)*. LNCS 5825, pp. 37–52. Springer, 2009.

[ABPV10] J. B. Almeida, M. Barbosa, J. S. Pinto, B. Vieira. Deductive Verification of Cryptographic Software. *To appear NASA Journal of Innovations in Systems and Software Engineering*, 2010.

[Bar09] M. Barbosa. CACE Deliverable D5.2: Formal Specification Language Definitions and Security Policy Extensions. 2009. Available from http://www.cace-project.eu.

[BBC+07] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In *International Symposium on Formal Methods for Components and Objects (FMCO)*. LNCS. Springer-Verlag, 2007.

[BED+06] M. Barnett, B. yuh Evan Chang, R. Deline, B. Jacobs, K. R. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO 2005)*. LNCS 4111, pp. 364–387. Springer-Verlag, 2006.

[BFM+08] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto. ACSL: ANSI/ISO C Specfication Language. CEA LIST and INRIA, 2008.

[BMF+] M. Barnett, P. Müller, M. Fähndrich, W. Schulte, K. Rustan, M. Leino, H. Venter. Specification and Verification: The Spec # Experience.

[CCK06] S. Conchon, E. Contejean, J. Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories. 2006. http://ergo.lri.fr/papers/ergo.ps.

[CDH+09] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies. VCC: A practical system for verifying concurrent C. In *Conf. Theorem Proving in Higher Order Logics*. LNCS 5674. Springer, 2009.

[DNS05] D. Detlefs, G. Nelson, J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM* 52(3):365–473, 2005.

[FLL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. In *ACM SIGPLAN Conference on Programming language design and implementation (PLDI'02)*. Pp. 234–245. ACM, 2002.

[FM04] J.-C. Filliâtre, C. Marché. Multi-prover Verification of C Programs. In Davies et al. (eds.), *ICFEM*. LNCS 3308, pp. 15–29. Springer, 2004.

[LRL+00] G. T. Leavens, C. Ruby, K. R. M. Leino, E. Poll, B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Proceedings of OOPSLA '00 (Poster session addendum)*. Pp. 105–106. ACM, New York, NY, USA, 2000.

[MB08] L. de Moura, N. Bjørner. *Z3: An Efficient SMT Solver*. LNCS 4963/2008, pp. 337–340. Springer Berlin, April 2008.

[MM10] C. Marché, Y. Moy. Jessie Plugin Tutorial. INRIA, 2010.

[MPU04] C. Marché, C. Paulin-mohring, X. Urbain. The Krakatoa Tool for Certication of Java/JavaCard Programs annotated in JML. 2004.

[The08] The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.2. 2008. http://coq.inria.fr.

A CAO implementation of AES

```
typedef GF2  := mod[ 2 ];
typedef GF2N := mod[ GF2<X>/X**8 + X**4 + X**3 + X + 1];
typedef GF2V := vector[8] of GF2;
typedef S,K  := matrix[4,4] of GF2N;
typedef Row  := matrix[1,4] of GF2N;
typedef Col  := matrix[4,1] of GF2N;
typedef RowV,ColV := vector[4] of GF2N;

def M : matrix[8,8] of GF2 := { ... };
def C : vector[8]   of GF2 := { ... };
def mix : matrix[4,4] of GF2N := { ... };

def SBox( e : GF2N ) : GF2N {
  def x : GF2N;
  if (e == [0]) { x := [0]; } else {   x := [1] / e; }
  def A : matrix[8,1] of GF2:=(matrix[8,1] of GF2)(GF2V)x;
  def B : GF2V := (GF2V)(M*A);
  return ((GF2N)B) + ((GF2N)C);
}
def SubBytes( s : S ) : S {
  def r : S;
  seq i := 0 to 3
    seq j := 0 to 3 {r[i,j] := SBox( s[i,j] );}
  return r;
}
def SubWord( w : vector[4] of GF2N ) :
  vector[4] of GF2N {
  def r : vector[4] of GF2N;
  seq i := 0 to 3 { r[i] := SBox( w[i] ); }
  return r;
}
def ShiftRows( s : S ) : S
{
  def r : S;
  seq i:= 0 to 3 {r[i,0..3]:=(Row)(((RowV)s[i,0..3])|>i);}
  return r;
}
def MixColumns( s : S ) : S
{
  def r : S;
  seq i := 0 to 3 { r[0..3,i] := mix * s[0..3,i]; }
  return r;
}
def AddRoundKey( s : S, k : K ) : S
{
  def r : S;
  seq i := 0 to 3
    seq j := 0 to 3 { r[i,j] := s[i,j] + k[i,j]; }
  return r;
}
def FullRound( s : S, k : K ) : S
{
    return MixColumns( ShiftRows( SubBytes(s) ) ) + k;
}
def Aes( s : S, keys : vector[11] of K) : S
{
  seq i := 1 to 9 { s := FullRound( s,keys[i] ); }
  return ShiftRows( SubBytes(s) ) + keys[10];
}
```