CAOVerif: An Open-Source Deductive Verification Platform for Cryptographic Software Implementations

José Bacelar Almeida[a,*], Manuel Barbosa[a,**], Jean-Christophe Filliâtre[b,*], Jorge Sousa Pinto[a,*], Bárbara Vieira[a,*]

[a]*HASLab – INESC TEC and Universidade do Minho, Braga, Portugal*
[b]*INRIA Saclay - Île-de-France, ProVal, Orsay, France*
LRI, Université Paris-Sud, CNRS, Orsay, France

Abstract

CAO is a domain-specific imperative language for cryptography, offering a rich mathematical type system and crypto-oriented language constructions. We describe the design and implementation of a deductive verification platform for CAO and demonstrate that the development time of such a complex verification tool could be greatly reduced by building on the Jessie plug-in included in the Frama-C framework. We discuss the interesting challenges raised by the domain-specific characteristics of CAO, and describe how we tackle these problems in our design. 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 Verification, Program Verification, Cryptographic Software, Deductive Verification

1. Introduction

The development of cryptographic software is clearly distinct from other areas of software engineering. The design and implementation of cryptographic software is inherently interdisciplinary, drawing 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 the challenges that they face. The formal

---

*Corresponding author
**Principal corresponding author

Email addresses: `jba@di.uminho.pt` (José Bacelar Almeida), `mbb@di.uminho.pt` (Manuel Barbosa), `Jean-Christophe.Filliatre@lri.fr` (Jean-Christophe Filliâtre), `jsp@di.uminho.pt` (Jorge Sousa Pinto), `barbarasv@di.uminho.pt` (Bárbara Vieira)

This work was supported by Project Best Case, co-financed by the North Portugal Regional Operational Programme (ON.2 – O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF).

verification tool described in this work has been developed to allow the static analysis of code written in CAO [1], a domain-specific language for cryptography. Our tool, which we call CAOVerif, was employed in the formal verification of an open-source library written in CAO and it has been tuned to enable the fully automatic verification of simple properties (e.g., safety properties), while still enabling the interactive validation of more ambitious proof goals.

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, usually brought to practice through the use of *contracts* – specifications annotated into the programs, consisting of preconditions, postconditions, invariants, frame conditions, and other elements. We use the expression *deductive verification* to distinguish this approach from other ways of checking properties of programs, such as *software model checking* [2, 3, 4]. While the theoretical principles of this form of verification have been established for decades now, only in recent years have verification tools evolved to cover features of realistic programming languages based on contracts (notably an adequate treatment of pointers and dynamic data structures). Such tools are available for instance for C [5, 6], C# [7] or Java [8].

In this work we build on Frama-C [6], an extensible open-source framework where static analyses of C programs are implemented by a series of plug-ins. Jessie [9] 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 Conditions Generator* (VCGen), which produces a set of proof obligations that can be discharged using a multitude of proof tools, including the Coq proof assistant [10], and the Simplify [11], Alt-Ergo [12], and Z3 [13] automatic theorem provers.

Motivation. Experience shows [14**?**] that a tool such as Frama-C has 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 *automated* verification. 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 (e.g., loop unrolling, single-iteration loops and break statements), intensive use of macros, etc. Our goal is to take advantage of the characteristics of CAO to construct a domain-specific verification tool, allowing for the same generic deductive verification techniques that can be applied over C implementations, but simplifying the verification of security-relevant properties and, hopefully, providing a higher degree of automation.

Contributions. We describe the design and implementation of CAOVerif, a deductive verification platform for CAO. We show that CAO presents interesting challenges for formal verification, concerning not only its rich type system, but also the cryptography-oriented language constructions that it offers. We describe how we tackle these challenges in our design, namely by presenting what we believe to be the first formalisation in first-order logic of the rich mathematical data types that are used in cryptography, for the purpose of deductive verification. For each CAO data type we present an axiomatic model that can be used by automatic proof tools to discharge various verification goals. However, the complex semantics of the CAO data types and the intricacy of our axiomatic models raises the natural question of whether our formalization is sound. We therefore describe how we have formally proven the soundness of our axiomatic models using the Coq proof assistant. Finally, we also demonstrate that the development time of such a complex verification tool can be greatly reduced by relying on the Jessie plug-in of the Frama-C framework. We base our presentation on real-world examples of CAO code, extracted from the open-source code of the NaCl cryptography library [15]. We show how we fine tuned CAOVerif to enable the fully automatic formal verification of simple properties (in particular safety properties), and also how more ambitious proof goals (arising in general proofs of functional correctness) can be addressed using interactive proofs.

This work is an extended version of [16]. In addition to an axpanded presentation of the design and implementation of CAOVerif, we include as supplementary material the formal verification of the axiomatic models used in the tool, as well as a real-world case study that allows us to demonstrate a wider range of features and capabilities of CAOVerif.

Organisation of the Paper. The next section expands on the application scenario and functional requirements for CAOVerif. 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 CAOVerif, including the generation of safety proof obligations. In Section 5 we show how we have proved the soundness of our axiomatic models in Coq and in Section 6 we describe an example of a real-world application of CAOVerif. Section 7 discusses related work and Section 8 has some concluding remarks.

2. Deductive verification of CAO programs

The CAO language. CAO allows for the practical description of cryptography kernels (e.g., block ciphers and hash functions) and sequences of algebraic operations (e.g., finite field arithmetics) for public-key cryptography. The language has been designed to allow the programmer to work over a syntax that is similar to that of C, whilst focusing on the aspects of cryptographic primitive implementation that are most critical for security and efficiency. A detailed specification of CAO can be found in [17, 18]. AppendixA includes source code extracted from a CAO implementation of the NaCl crypography library.

As a C-like language, CAO includes conditionals and loops, as well as global variable declarations, function declarations and procedures. The seq statement permits expressing loop constructions where the number of iterations can be statically determined. Here, the iterator is an integer variable, seen as read-only within the loop body. The memory model of CAO is extremely simple (there is no dynamic memory allocation, evaluation of expressions produces no side effects, and the language has a call-by-value semantics). Furthermore, CAO does not support any input/output constructions, as it is targeted at implementing the core components in cryptographic libraries.

The syntax of expressions is also similar to that of C, but the native types and operators in the language are highly expressive and tuned to the specific domain of cryptography. The CAO type system includes a set of primitive types: int for arbitrary precision integers, bits[n] for bit strings of finite length, mod[n] for rings of residue classes modulo an integer (intuitively, arithmetic modulo a composite integer, or a finite field of order n if the modulus is prime) and bool for boolean values. Derived types include the product construction struct, the generic one-dimensional container vector[n] of τ, the algebraic notion of matrix matrix[i,j] of τ, and the construction of an extension to a finite field τ using a polynomial $p(X)$, denoted mod[τ <X> / $p(X)$].[2]

Algebraic operators are overloaded so that expressions can include integer, ring/finite-field and matrix operations. The ** operator represents the exponentiation operation, where the basis can be an integer or a value in a modular type, and the exponent must be a non-negative integer. 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 the range selection (..) operator. Vectors and bit strings can be concatenated using the @ operator.

An implementation of a type-checker for CAO programs has been derived from the CAO type system formalisation [18]. Hence, we assume that the input CAO program has been type checked and that CAOVerif has access to an Abstract Syntax Tree (AST) annotated with the resulting type information. We remark 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.s The soundness of this type system has recently been established with respect to the semantics of CAO [19]. This result implies that a correctly typed CAO program can only give rise to a well-defined set of *trapped* errors. This notion has a direct bearing on the discussion of what *safety* means for CAO programs.

Safety in CAO. Checking 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 *safety verification*. This type of verification goal is admittedly a modest one; nevertheless, not only is it a critical aspect for practical applications, but it is also frequently a challenge for existing tools and may become a labour-intensive

[2]Semantics of an extension field of order n^d is defined as arithmetic modulo an irreducible polynomial $p(X)$ with coefficients in τ, where τ is a field of order n and $p(X)$ has degree d.

activity. A requirement for CAOVerif is that safety verification should involve little or no intervention from the end-user.

Program safety in CAO has two dimensions: *memory safety* and *safety of arithmetic operations*. A memory-safe program never fails at run-time 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 dedicated verification tools are often needed for this task. However, for correctly typed 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.

The safety of arithmetic operations is more interesting. In CAO we have four algebraic types: arbitrary precision integers, rings of residue classes modulo a composite number, 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 certain inputs, and that such pathological cases might not be caught during type-checking.

Assuming a CAO program correctly type-checks, then matrix addition and multiplication are intrinsically safe. The safety of integer operations includes the classic division-by-zero condition. Furthermore, the exponentiation operator over integers, rings and finite-fields is only defined for non-negative exponents. Rings and finite fields pose specific interesting problems, as they are not syntactically distinct CAO types. Take the following declarations.[3]

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

All these operations are safe, except the initialization of d. This is because the multiplicative inverse modulo 10 is defined only for integers that are coprime with 10. This means that, whenever a division occurs in the mod[n] type, one must also ensure that the divisor is coprime 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. 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 base type represents a finite field (which might not be the case for the mod[n] type) but also that the provided polynomial is *irreducible*.

Extending CAO with annotations. Contract-based program verification requires the use of an interface specification language to write contracts and other annotations embedded in the programs to be verified. CAO-SL is a specification

[3]Here the [·] syntax on literals distinguishes literals of modular types from integer literals.

language that can be used in annotations over CAO programs, to express behavioral properties of these programs. These annotations are embedded in comments (and thus ignored by the CAO compiler) using a special format recognised by CAOVerif. CAO-SL is strongly inspired by ACSL [6]; it stands to CAO in the same way that ACSL stands to C. The logical expressions used in annotations correspond to CAO expressions with additional constructs. CAO-SL includes the definition of function contracts with pre- and postconditions, statement annotations (e.g., assertions and loop variants/invariants), and other common annotations. 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 [1]. In this paper, various features of this language will be introduced gradually, as we describe the CAOVerif architecture and implementation.

3. Implementation of CAOVerif

CAOVerif follows the same approach used in other scenarios for general-purpose languages such as Java [20] and C [21]. The CAOVerif architecture relies on the Jessie plug-in, which itself uses Why as a back-end and is one of the components integrated into the Frama-C framework. This allowed us to significantly reduce development time and effort.

Using Jessie as a back-end. 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 Jessie input language is a simply typed imperative language with a precisely defined semantics. Programmers are not expected to produce Jessie source programs from scratch: Jessie is used as an intermediate language, for instance for verification of C programs in the Frama-C framework. The language was developed in parallel with ACSL, and they share many constructions. The language combines operational and logical 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 consists of 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.

Implementation strategy. A detailed explanation of the implementation strategy that we adopted in CAOVerif can be found in [16]. In a nutshell, in addition to the translated CAO program in the Jessie input language, CAOVerif generates a Jessie prelude that includes a model in first order logic of the CAO types used in the program. More precisely, for each CAO type, CAOVerif generates a theory, including the definition of logical functions together with axioms to model their behavior. Some lemmas and predicates may also be introduced to facilitate the process of proving verification goals. Intuitively, the generated Jessie program should enjoy two types of properties. Firstly, it 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. For this, the models must describe the properties of the object types as completely as possible. Secondly, the translation should be *sound*: the Jessie model should not allow proving assertions about CAO data structures that are not valid according to the language semantics.

Emphasis on Automation. The fact that Jessie relies on Why as a VCGen, which is a *multi-prover* tool, means that it is possible to export verification conditions (VC) 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 handle the remaining conditions. Our translation enables varying degrees of automation, depending on the complexity of the verification goals. As is the case with VCGens for other realistic languages, one expects safety conditions to be proved with a high degree of automation, whereas a lower degree is acceptable for other 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 or integers). Finally, all models can be further refined into Coq models, since interactive proof may be the last resort for discharging first-order VCs.

The degree of automation that we can achieve in verifying the safety of CAO programs is quite high. For example, once the code has been suitably annotated, we are able to carry out the safety verification of the entire CAO implementation of AppendixA without user intervention, and similarly for other examples[4] that combine finite field, vector and matrix operations across several (not necessarily leaf) functions. We are also able to deal with more ambitious proof goals, such as those for functional correctness in this case study, with only minor intervention from the user in interactively discharging proof obligations.

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 CAOVerif, which essentially correspond to the rich cryptography-specific data types that are available in

[4]One such example is the AES implementation from which we extract various snippets that we include for illustrative purposes in the next section.

Figure 1: Type translation.

$\lceil \mathsf{int} \rceil = \mathsf{integer}$	$\lceil \mathsf{bits}[n] \rceil = \mathsf{bits}$
$\lceil \mathsf{bool} \rceil = \mathsf{boolean}$	$\lceil \mathsf{matrix}[n_1{,}n_2]\ \mathsf{of}\ \tau \rceil = \mathsf{matrix\_}\lceil \tau \rceil$
$\lceil \mathsf{void} \rceil = \mathsf{unit}$	$\lceil \mathsf{mod}[\tau <X> / \ p(X)] \rceil = \mathsf{field\_}\lceil \tau \rceil\mathsf{\_}\lceil p(X) \rceil$
$\lceil \mathsf{mod\_}n(p) \rceil = \mathsf{mod\_}p$	$\lceil \mathsf{vector}[n_1]\ \mathsf{of}\ \tau \rceil = \mathsf{vector\_}\lceil \tau \rceil$

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, which is also direct. In the following, $\lceil x \rceil$ denotes the translation of a part of the input CAO program x into Jessie. Here x can denote any part of the input AST, e.g., a full program, a type declaration, an expression, etc.

Figure 1 gives an overview of how CAO type declarations are translated into Jessie type declarations. Some CAO primitive types are translated to Jessie primitive types, namely int, bool and void. This means that, for these CAO data types, we directly benefit from the models already provided by the Jessie plug-in for reasoning about the target Jessie native types.

The remaining CAO types are mapped into newly declared Jessie logic types. Note that, for parametrised data types such as mod[n], the target type in Jessie is named so as to explicitly capture the type parameter. This also explains why we use the translation operation recursively in Figure 1. In the following, we discuss how we enrich the generated Jessie input file with logic models that partially capture the semantics of the translated CAO types, in order to enable both automatic and interactive reasoning about the input CAO program.

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 two dimensions in the case of matrices, and that we set Jessie type bool as the content type in the case of bits.

Additionally, CAO includes elaborate operators to deal with these container types that are fine-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 a CAO implementation of the Advanced Encryption Standard (AES) block cipher.

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

What we have here is a sequence of rotation (|>) operations applied to the i^{th} row of a 4×4 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

Figure 2: Declarations and axioms for vector types.

$$\begin{array}{l}
\mathsf{blit\_vector\_}\lceil\tau\rceil \;:\; \mathsf{vector\_}\lceil\tau\rceil \to \; \mathsf{vector\_}\lceil\tau\rceil \to \mathsf{integer} \to \mathsf{integer} \to \; \mathsf{vector\_}\lceil\tau\rceil \\
\mathsf{shift\_vector\_}\lceil\tau\rceil \;:\; \mathsf{vector\_}\lceil\tau\rceil \to \mathsf{integer} \to \mathsf{vector\_}\lceil\tau\rceil
\end{array}$$

$$\forall v, \mathit{ofs}, i.\; \mathsf{get\_vector\_}\lceil\tau\rceil(\mathsf{shift\_vector\_}\lceil\tau\rceil(v, \mathit{ofs}), i) = \mathsf{get\_vector\_}\lceil\tau\rceil(v, (\mathit{ofs} + i))$$

$$\begin{array}{l}
\forall \mathit{src}, \mathit{dest}, \mathit{ofs}, \mathit{len}, i.\; \mathit{ofs} \leq i < (\mathit{ofs} + \mathit{len}) \implies \\
\quad \mathsf{get\_vector\_}\lceil\tau\rceil(\mathsf{blit\_vector\_}\lceil\tau\rceil(\mathit{src}, \mathit{dst}, \mathit{ofs}, \mathit{len}), i) = \mathsf{get\_vector\_}\lceil\tau\rceil(\mathit{src}, i - \mathit{ofs})
\end{array}$$

$$\begin{array}{l}
\forall \mathit{src}, \mathit{dest}, \mathit{ofs}, \mathit{len}, i.\; i < \mathit{ofs} \;\vee\; i \geq (\mathit{ofs} + \mathit{len}) \implies \\
\quad \mathsf{get\_vector\_}\lceil\tau\rceil(\mathsf{blit\_vector\_}\lceil\tau\rceil(\mathit{src}, \mathit{dst}, \mathit{ofs}, \mathit{len}), i) = \mathsf{get\_vector\_}\lceil\tau\rceil(\mathit{dst}, i)
\end{array}$$

dimensions. Here, this operator is used to select an entire row in the matrix, which is cast into the correct vector type (here the RowV type denotes a vector of size 4) 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 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 4.5.

Intuitively, the shift_vector logic function takes as input a vector of arbitrary length, starting in position 0, and produces the vector that starts at position i. The blit_vector 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 2.

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

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

where i and j are integer expressions. We remark that although the translation disregards the upper bound j in the call to shift_vector, the type-checking phase has ensured that the range selection operation $\mu[i..j]$ with μ of type vector[n] of τ, returns type vector[$j - i + 1$] of τ, thus implicitly taking that upper bound into account. Furthermore, all future accesses to the resulting vector will be checked for safety within the valid bounds prescribed by the correct data type.

Range assignment. Assigning to a region in a vector is modeled directly using the $\mathsf{blit\_vector}$ function.

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

Here, $=$ denotes assignment in Jessie.

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

$$\lceil \mu_1 \ @ \ \mu_2 \ \rceil \equiv \mathsf{blit\_vector}(\lceil \mu_2 \rceil, \lceil \mu_1 \rceil, n_1, n_2)$$

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 $\mathsf{blit\_vector}$ function. The rotate operations are commonly known as circular shifts. As an example, consider the bits literal: $\mathsf{0b1101001}$. 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 an upwards (resp. downwards) rotate corresponds to the intuitive interpretation of a left (resp. right) rotation. An example of a down rotate is therefore $0b1101001 \ |\!> 3 \ = \ 0b\underline{001}1101$ and an example of an up rotate is $0b1101001 \ <\!| \ 3 \ = \ 0b1001\underline{110}$. In our model, for a CAO expression e of type $\mathsf{vector}[n]$ of τ or $\mathsf{bits}[n]$, we have:

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

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 $\mathsf{blit\_vector}$ function.

Logical shifts are handled in a similar way, but resorting to $\mathsf{bits\_null\_vector}$ (a logical variable representing the all-zeroes bits value) to fill in the positions left vacant by the operation, i.e.,

$$\begin{aligned}\lceil e \ \ll \ i \rceil \ &\equiv \ \mathsf{blit\_vector}(\mathsf{shift\_vector}(e, 0), \mathsf{bits\_null\_vector}, i, n-i)\\ \lceil e \ \gg \ i \rceil \ &\equiv \ \mathsf{blit\_vector}(\mathsf{bits\_null\_vector}, \mathsf{shift\_vector}(\lceil e \rceil, i), n-i, i)\end{aligned}$$

To model the behaviour of the $\mathsf{bits\_null\_vector}$ logical variable we include the following axiom in the bits type theory:

$$\forall \ \mathsf{integer} \ j. \ \mathsf{bits\_get}(\mathsf{bits\_null\_vector}, j) == \mathsf{false};$$

We remark that our model of the operations over bit strings is complete, and therefore allows us to deduce the natural properties of bit string operations. Furthermore, surprisingly complex properties can be derived automatically. Consider, for example, the bistring rotation operation and the property that rotating n times a bit string of length n in the same direction yields the original bit string:

$$\begin{aligned} \forall i.\ \ 0 \leq i < n \implies \lceil e[i] \rceil == \lceil (e \mathbin{|>} n)[i] \rceil \\ \forall i.\ \ 0 \leq i < n \implies \lceil e[i] \rceil == \lceil (e \mathbin{<|} n)[i] \rceil \end{aligned}$$

Or, more generally, for a bit string of length $n_1 + n_2$,

$$\forall i.\ \ 0 \leq i < (n_1 + n_2) \implies \lceil (e \mathbin{<|} n_1)[i] \rceil == \lceil (e \mathbin{|>} n_2)[i] \rceil$$

Our model enables proving these properties automatically using, e.g., Alt-Ergo.

Matrices. Our model of matrices is 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

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

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

Axiom (Matrix addition)**.** *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 the mathematical (iterative) sequence summation operator Σ.

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

$$\begin{aligned} \lceil\ \mu_1\ +\ \mu_2\ \rceil &= \mathsf{matrix\_}\lceil\tau\rceil\mathsf{\_add}(\lceil\mu_1\rceil, \lceil\mu_2\rceil) \\ \lceil\ \mu_1\ *\ \mu_2\ \rceil &= \mathsf{matrix\_}\lceil\tau\rceil\mathsf{\_mult}(\lceil\mu_1\rceil, \lceil\mu_2\rceil). \end{aligned}$$

Bitwise operations. We complete this section with a brief description of how bit-wise 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 bit-wise operations on bits is based on the following logic functions, which are axiomatized in the obvious way:

$$\begin{array}{rl} \mathsf{bits\_bitwise\_xor} : \mathsf{bits} \to \mathsf{bits} \to \mathsf{bits} & \mathsf{bits\_bitwise\_and} : \mathsf{bits} \to \mathsf{bits} \to \mathsf{bits} \\ \mathsf{bits\_bitwise\_or} : \mathsf{bits} \to \mathsf{bits} \to \mathsf{bits} & \mathsf{bits\_bitwise\_neg} : \mathsf{bits} \to \mathsf{bits} \end{array}$$

CAO bit-wise operations are translated as:

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

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

4.2. Rings, fields and extension fields

Residue classes modulo n. The $\mathsf{mod}[n]$ 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 $\mathsf{mod}[n]$ permits programmers to take full advantage of the fact that $\mathbb{Z}_n$ is a field.

More in detail, the Jessie model for the $\mathsf{mod}[n]$ type is based on the congruence relation defined by n over the integers. For a positive integer n, two integers a and b are said to be *congruent modulo n* if $a - b$ is an integer multiple of n, and this is denoted by $a \equiv b\ (mod\ n)$.

For any integer a, the corresponding equivalence class modulo n is denoted by $[a]$, and it corresponds to the set $a + n\mathbb{Z}$, where $n\mathbb{Z}$ (the set of multiples of n). For all integers a, the unique value r satisfying $a = nq + r \ \wedge \ 0 \leq r < n$ (for some integer q) is called the *least residue* of a modulo n. The set $\{0, 1, ..., n-1\}$ is therefore called the set of least residues modulo n. Each residue class modulo n is represented by a least residue modulo n.

The model of $\mathsf{mod}[n]$ starts with the definition of the logic type $\mathsf{mod\_}n$, which intuitively is inhabited by the residue classes modulo n. This type is equipped with logic functions that convert to and from the Jessie integer type, as well as the mapping that results from their composition.

$$\begin{array}{r} \mathsf{int\_of\_mod\_}n : \mathsf{mod\_}n \to \mathsf{integer} \\ \mathsf{mod\_}n\mathsf{\_of\_int} : \mathsf{integer} \to \mathsf{mod\_}n \\ \mathsf{mod\_}n : \mathsf{integer} \to \mathsf{integer} \end{array}$$

The conversion to integers captures the homomorphism mapping a residue class into the corresponding least residue, whereas the converse operation represents the homomorphism mapping an integer into its residue class. The $\mathsf{mod\_}n$ function represents the composition of the previous two, and associates to each $a \in \mathbb{Z}$ the least residue $r \in \mathbb{Z}$ of $[a]$. The model includes a set of axioms for the following mathematical properties of these functions:

$$\begin{array}{l} \forall x.\ 0 \leq \mathsf{int\_of\_mod\_}n(x) \leq n - 1 \\ \forall x.\ 0 \leq x \leq n - 1 \implies \mathsf{mod\_}n(x) = x \\ \forall x.\ x \geq n \implies \mathsf{mod\_}n(x) = \mathsf{mod\_}n(x - n) \\ \forall x.\ x < 0 \implies \mathsf{mod\_}n(x) = \mathsf{mod\_}n(x + n) \\ \forall x.\ \mathsf{mod\_}n(\mathsf{int\_of\_mod\_}n(\mathsf{mod\_}n\mathsf{\_of\_int}(x))) = \mathsf{mod\_}n(x) \end{array}$$

Equipped with these functions we can base our entire model of integers modulo n on the theory of integers included in Jessie, which permits taking advantage of built-in arithmetic supported by many automatic provers.

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, which is then carried out over the integers. Finally, we apply mod_n_of_int to the result to recover the equivalence class that represents the result. Hence, the translation of arithmetic operations on type mod[n] is given as follows, for $op \in \{+, -, *\}$.

$$\begin{aligned}
\lceil e_1 \ op \ e_2 \rceil &\equiv \mathsf{mod\_}n\mathsf{\_of\_int}(\mathsf{int\_of\_mod\_}n(\lceil e_1 \rceil) \ op_{\mathsf{integer}} \ \mathsf{int\_of\_mod\_}n(\lceil e_2 \rceil)) \\
\lceil e_1 \ ** \ e_2 \rceil &\equiv \textbf{let } x = \lceil e_2 \rceil \textbf{ in assert } x \geq 0; \\
&\quad\ \mathsf{mod\_}n\mathsf{\_of\_int}(\mathsf{int\_of\_mod\_}n(\lceil e_1 \rceil) **_{\mathsf{integer}} \ x) \\
\lceil e_1 \ / \ e_2 \rceil &\equiv \textbf{let } x = \mathsf{int\_of\_mod\_}n(\lceil e_2 \rceil) \textbf{ in assert } \mathsf{gcd}(x, n) = 1; \\
&\quad\ \mathsf{mod\_}n\mathsf{\_of\_int}(\mathsf{int\_of\_mod\_}n(\lceil e_1 \rceil) *_{\mathsf{integer}} \mathsf{inv\_mod}(x, n))
\end{aligned}$$

Exponentiation is translated so as to ensure that verification guarantees that the exponent is nonnegative, which would otherwise result in an error according to the semantics of the language. Also 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 automatically discharge some proof obligations, are axiomatized as:

$$\begin{aligned}
&\forall x. \ \mathsf{gcd}(\mathsf{int\_of\_mod\_}n(x), n) = 1 \implies \\
&\qquad \mathsf{mod\_}n(\mathsf{int\_of\_mod\_}n(x) *_{\mathsf{integer}} \mathsf{inv\_mod}(\mathsf{int\_of\_mod\_}n(x), n)) = \mathsf{mod\_}n(1) \\
&\forall x, y. \ \mathsf{mod\_}n(\mathsf{int\_of\_mod\_}n(x) *_{\mathsf{integer}} y) = \mathsf{mod\_}n(1) \implies \\
&\qquad \mathsf{inv\_mod}(\mathsf{int\_of\_mod\_}n(x), n) = \mathsf{mod\_}n(y)
\end{aligned}$$

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.

$$\forall x, n. \quad \mathsf{is\_prime}(n) \ \land \ (0 < x < n) \implies \mathsf{gcd}(x, n) = 1$$

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 of 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 same AES implementation referred above:

```
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 this form are also algebraic types that model the finite field of order n^d where n is a prime number and d is the degree of the irreducible polynomial $p(X)$. In other words, type declarations such as this are only valid when n is prime and $p(X)$ is irreducible. In CAO, each such type represents a specific construction of an extension field represented over the polynomial ring $\mathbb{Z}_n[X]$. The semantics of the algebraic operations over such types are defined based on polynomial arithmetics modulo $p(X)$.

The theory of extension fields of this form begins with the definition of a logic type $\mathsf{ring\_mod\_}n$ that represents the ring of polynomials over the base type $\mathsf{mod}[n]$. Arithmetic operations over the polynomial ring are not included in the model, as they do not exist in CAO. However, the following two logic functions are included to allow constructing elements in the ring.

$$\begin{aligned}\mathsf{ring\_mod\_}n\mathsf{\_monomial} &: \mathsf{mod\_}n \rightarrow \mathsf{integer} \rightarrow \mathsf{ring\_mod\_}n\\ \mathsf{ring\_mod\_}n\mathsf{\_add} &: \mathsf{ring\_mod\_}n \rightarrow \mathsf{ring\_mod\_}n \rightarrow \mathsf{ring\_mod\_}n\end{aligned}$$

A monomial can be represented by its coefficient (which is an element of $\mathsf{mod}[n]$) and its degree (an integer). A polynomial can be defined as an addition of monomials. In this way, the CAO literal that corresponds to the irreducible polynomial $p(X)$ can be represented in our logic model. Although we do not take advantage of this representation in our axiomatic models, it is essential if one intends to use an interactive theorem prover to discharge proof obligations that require a complete formalization of the representation of the extension field.

The central part of the model are the definitions for type $\mathsf{field\_mod\_}n\mathsf{\_poly\_}p(x)$ and the corresponding arithmetic operations. The Jessie translation of the arithmetic operations defined for type $\mathsf{mod}[\mathsf{mod}[n] <X> / \ p(X)]$ is then a direct one:

$$\begin{aligned}\lceil e_1 \ op \ e_2\rceil &\equiv \lceil e_1\rceil \ op_{\mathsf{field\_mod\_}n\mathsf{\_poly\_}p(x)} \lceil e_2\rceil\\ \lceil e_1 \ {**} \ e_2\rceil &\equiv \textbf{let } x = \lceil e_2\rceil \textbf{ in assert } x \geq 0;\\ &\qquad \lceil e_1\rceil \ {**}_{\mathsf{field\_mod\_}n\mathsf{\_poly\_}p(X)} \ x\\ \lceil e_1 \ / \ e_2\rceil &\equiv \textbf{let } x = \lceil e_2\rceil \textbf{ in assert } x \neq 0_{\mathsf{field\_mod\_}n\mathsf{\_poly\_}p(X)};\\ &\qquad \lceil e_1\rceil \ div_{\mathsf{field\_mod\_}n\mathsf{\_poly\_}p(X)} \ x\end{aligned}$$

where $op \in \{+, -, *\}$. There are also special cases for exponentiation and division, ensuring that safety proof obligations are generated to check if the integer exponent is nonnegative and that the divisor is different from zero, respectively.

The model also includes a set of axioms that aim to increase the degree of automation provided by CAOVerif. The idea is the following. There is no integrated support for this sort of mathematical type in the automatic provers interfaced with Jessie, and so one can have no hope of dealing automatically with complex proof obligations over such types. However, some simple properties such as cancellation rules can still be captured in first-order logic in order to deal with more trivial steps. The following axioms, where F stands for $\mathsf{field\_mod\_}n\mathsf{\_poly\_}p(X)$, capture precisely this type of property.

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

Finally, literals of the extension field types are modeled in Jessie as vectors of polynomial coefficients. Therefore, the model also includes logic functions to access and update these coefficients, together with the usual two axioms for the theory of arrays.

$$\begin{aligned}\mathsf{field\_mod\_}n\mathsf{\_poly\_}p(x)\mathsf{\_get\_coef} &: \mathsf{field\_mod\_}n\mathsf{\_poly\_}p(x) \to \mathsf{integer} \to \mathsf{mod\_}n\\ \mathsf{field\_mod\_}n\mathsf{\_poly\_}p(x)\mathsf{\_set\_coef} &: \mathsf{field\_mod\_}n\mathsf{\_poly\_}p(x) \to \mathsf{integer} \to \mathsf{mod\_}n\\ &\qquad \to \mathsf{field\_mod\_}n\mathsf{\_poly\_}p(x)\end{aligned}$$

The null polynomial is represented by logical variable $\mathsf{field\_}\lceil\tau\rceil\mathsf{\_poly\_}p(x)\mathsf{\_zero}$. An auxiliary axiom states that all of its coefficients are the zero element in τ.

As an example of where our theory can be useful for automation purposes, consider the following snippet of an AES implementation in CAO.

```
def SBox( e : GF2N ) : GF2N {
  def x : GF2N;
  if (e == [0]) { x := [0]; } else {   x := [1] / e; }
  ...
```

The safety of this construction relies on the fact that the division in the else branch of the if statement is only executed if the input parameter is not the zero element in the field. Our models allow an automatic prover to validate that the equality comparison with literal [0] in the condition of the if statement actually implies that this is the case.

Returning to the example type declarations 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, with the single caveat that the correct base type must be considered when defining the ring of polynomials over which the extension field elements are represented.

4.3. Structs

As in C, structs in CAO are structured types which aggregate a fixed number of fields, possibly of different types, into a single type. Typically, the struct type operations are access and update to struct fields, hence the Jessie model for the CAO structs is very similar to the vectors model. To access and update each field $\mathsf{field}_i : \tau_i$, the following two logic functions are declared:

$$\begin{aligned}\mathsf{struct\_}\lceil\tau\rceil\mathsf{\_get\_field}_i &: \mathsf{struct\_}\lceil\tau\rceil \to \lceil\tau_i\rceil\\ \mathsf{struct\_}\lceil\tau\rceil\mathsf{\_set\_field}_i &: \mathsf{struct\_}\lceil\tau\rceil \to \lceil\tau_i\rceil \to \mathsf{struct\_}\lceil\tau\rceil\end{aligned}$$

Figure 3: Casts ($\rightarrow$) and coercions ($\Rightarrow$)

$$\begin{array}{rr}
\mathsf{bits}[n] \Rightarrow \mathsf{int} & \mathsf{mod}[\tau <X> / \; p(X)] \rightarrow \mathsf{vector}[n] \text{ of } \tau \\
\mathsf{mod}[n] \rightarrow \mathsf{int} & \mathsf{matrix}[1{,}n] \text{ of } \tau \; of \; \tau \rightarrow \mathsf{vector}[n] \text{ of } \tau \\
\mathsf{int} \rightarrow \mathsf{bits}[n] & \mathsf{matrix}[n{,}1] \text{ of } \tau \rightarrow \mathsf{vector}[n] \text{ of } \tau \\
\tau \Rightarrow \mathsf{mod}[\tau <X> / \; p(X)] & \mathsf{vector}[n] \text{ of } \tau \rightarrow \mathsf{matrix}[1{,}n] \text{ of } \tau \\
\mathsf{vector}[n] \text{ of } \tau \rightarrow \mathsf{mod}[\tau <X> / \; p(X)] & \mathsf{vector}[n] \text{ of } \tau \rightarrow \mathsf{matrix}[n{,}1] \text{ of } \tau
\end{array}$$

The behavior of these functions is axiomatized as expected, although it is slightly more verbose in order to deal with the fact that our index into the structure is now an identifier rather than an integer.

4.4. 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 3 presents the allowed cast ($\rightarrow$) and coercion ($\Rightarrow$) operations between CAO types. Several examples of how these casts are used in CAO programs can be found in the use case included in AppendixA, e.g., in function crypto_scalarmult the value returned by function curve25519, a finite field element represented as a value of type mod, is cast into the int type before being cast into a bit string of appropriate site. 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:

$$\begin{array}{rcl}
e :: \mathsf{mod}[n] & \Longrightarrow & \lceil(\mathsf{int})\; e\rceil \;=\; \mathsf{int\_of\_mod\_}n(\lceil e\rceil) \\
e :: \mathsf{int} & \Longrightarrow & \lceil(\mathsf{mod}[n])\; e\rceil \;=\; \mathsf{mod\_}n\mathsf{\_of\_int}(\lceil e\rceil) \\
e :: \mathsf{int} & \Longrightarrow & \lceil(\mathsf{bits}[n])\; e\rceil \;=\; \mathsf{bits\_of\_int}(\lceil e\rceil) \\
e :: \tau & \Longrightarrow & \lceil(\mathsf{mod}[\tau \;<X> / \; p(X)])\; e\rceil = \\
 & & \quad \mathsf{field\_}\lceil\tau\rceil\mathsf{\_poly\_}p(x)\mathsf{\_set\_coef}(\mathsf{field\_}\lceil\tau\rceil\mathsf{\_poly\_}p(x)\mathsf{\_zero}, 0, \lceil e\rceil)
\end{array}$$

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 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

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

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**
int	e_1/e_2	$e_2 \neq 0$	×
	$e_1 \ ** \ e_2$	$e_2 \geq 0$	
mod_n(n)	e_1/e_2	gcd(int_of_mod_$n(e_2), n) = 1 \wedge$ int_of_mod_$n(e_2) \neq 0$	×
	$e_1 \ ** \ e_2$	$e_2 \geq 0$	
mod[τ $<X>$ / $p(X)$]	$e_1 \ / \ e_2$	$e_2 \neq 0$	
vector[n] of τ	$v[e]$	$0 \leq \lceil e \rceil < n$	
	$v \vert> i$, $v <\vert i$	$0 \leq \lceil i \rceil < n$	
	$v[i..j]$	$0 \leq \lceil i \rceil < n \wedge 0 \leq \lceil j \rceil < n \wedge \lceil i \rceil < \lceil j \rceil$	
matrix[n_1,n_2] of τ	$m[e_1, e_2]$	$0 \leq \lceil e_1 \rceil < n_1 \ \wedge \ 0 \leq \lceil e_2 \rceil < n_2$	
	$m[i..j, k..l]$	$0 \leq \lceil i \rceil < n_1 \ \wedge \ 0 \leq \lceil j \rceil < n_1 \ \wedge \ 0 \leq \lceil k \rceil < n_2 \ \wedge \ 0 \leq \lceil l \rceil < n_2 \ \wedge \ \lceil i \rceil < \lceil j \rceil \ \wedge \ \lceil k \rceil < \lceil l \rceil$	
bits[n]	$b[e]$	$0 \leq \lceil e \rceil < n$	
	$b \vert> i$, $b <\vert i$	$0 \leq \lceil i \rceil < n$	
	$b \gg i$, $b \ll i$	$0 \leq \lceil i \rceil < n$	

4.5. Automatic safety proof obligations

Following the same approach adopted in tools such as Frama-C, the CAO to Jessie translation in CAOVerif 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 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 exponentiation and 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.

To support the automatic verification of safety proof obligations, CAOVerif also enriches the translated Jessie code with lemmas that capture some number theoretic assumptions that are implicit in the type checking procedure. We believe that this approach is also useful in raising the programmer's awareness as to the necessity to ensure that these assumptions are true. Concretely, when an extension field is declared, CAOVerif automatically generates lemmas that capture the necessary conditions for these declarations to be meaningful according to the CAO semantics. For extensions mod[mod[n] $<X>$ / $p(X)$], CAOVerif generates lemmas for the following two predicates:

is_prime(n)   ring_mod_n_is_irreducible(field_mod_n_poly_$p(x)$_generator)

When the base type for the extension is already an extension field, only the irreducibility lemma is generated. Lemmas can be immediately used in proofs, e.g., the first lemma above can be used as an hypothesis in all proof obligations related to division operations in $\mathsf{mod}[n]$ requiring that the divisor is relative prime to the modulus. Note, however, that the presence of lemmas also originates new proof obligations corresponding to the validation of the lemmas themselves.

5. Proving the soundness of axiomatic models for CAO types

The correctness of CAOVerif depends critically on the soundness of the axiomatic models introduced in the previous section, with respect to the semantics of CAO types. In this section we describe how we used the Coq interactive theorem prover to formally establish the soundness of these models. More precisely, we describe how 1) we formalized the semantics of the CAO native operations for which CAOVerif includes an axiomatic model; and 2) we interactively proved the validity of the axioms included in our models with respect to the aforementioned semantics. We start by briefly explaining how we structured the Coq library supporting our proof.

Overview of the Coq library. For each of the CAO types we have created a Coq theory containing a matching type definition in Coq that establishes an interpretation domain for values of that type (with appropriate type parameterization in the case of modular and container types). Furthermore, for each CAO native operation that is modeled using first-order logic in CAOVerif, there exists a corresponding Coq function that is defined to formalize the correct semantics of the CAO operation. We also include additional Coq function definitions for the auxiliary logic functions used by CAOVerif (e.g., the shift and blit functions used in our models for containers). Finally, for each axiom in our Jessie models, we state a corresponding lemma in Coq. This includes, not only the explicit axioms we have presented in the previous section, but also additional lemmas to account for equivalences that are implicitly assumed in the translation carried out by CAOVerif. Recall that, for performance reasons, there is not a one-to-one matching between CAO native operations and Jessie logic functions. Instead, some operations are translated as a combination of calls to logic functions. For example, addition in type $\mathsf{mod}[n]$ is translated as

$$\mathsf{mod\_}n\mathsf{\_of\_int}(\mathsf{int\_of\_mod\_}n(e_1)\ +_{\mathsf{integer}}\ \mathsf{int\_of\_mod\_}n(e_2))\ ,$$

which implicitly relies on the following axiom (for all $n \geq 2$):

$$\begin{aligned}&\forall v_1, v_2\,.\ \ v_1\ +_{\mathsf{mod\_}n}\ v_2 = \\ &\qquad \mathsf{mod\_}n\mathsf{\_of\_int}(\mathsf{int\_of\_mod\_}n(v_1)\ +_{\mathsf{integer}}\ \mathsf{int\_of\_mod\_}n(v_2))\ .\end{aligned}$$

Similarly, vector concatenation is translated as $\mathsf{blit\_vector}(\mu_2, \mu_1, n_1, n_2)$, which implicitly relies on the following axiom:

$$\begin{aligned}&\forall v_1, v_2, len_1, len_2, i.\ \ 0 \leq i < len_1 + len_2 \implies \\ &\qquad \mathsf{get\_vector}(v_1\ @\ v_2, i) = \mathsf{get\_vector}(\mathsf{blit\_vector}(v_2, v_1, len_1, len_2), i)\ .\end{aligned}$$

Such axioms appear explicitly as lemmas in our Coq scripts.

Formalizing the interpretation domains of CAO types. To capture the semantics of CAO types, we used SSReflect (version 1.3pl4),[5] which includes formalizations of all mathematical structures existing in the CAO type system. Technically, we first defined an interpretation domain for each type, i.e., we associated each CAO type (now translated into a Coq definition) to a suitable mathematical type in SSReflect. Our mapping from CAO types to SSReflect types is as follows:

- bool is mapped as the Coq standard bool type.
- int is mapped to the Coq standard Z type.[6]
- bits[n] is mapped to Boolean tuples n.-tuple bool, i.e. sequences of a fixed given size.
- mod[n], for arbitrary $n \geq 2$, is mapped to the $'\mathsf{I}_n$ type of the zmodp library, corresponding to the ring $\mathbb{Z}_n$.
- mod[τ <X>/p(X)] is mapped to the finFieldType, which represents an abstract finite field in the finalg library.
- vector[n] of τ is mapped to tuples n.-tuple τ.
- matrix[n_1,n_2] of τ is mapped to the $\mathsf{M}[\tau]_{(n_1,n_2)}$ type of the matrix library.

An important aspect of this mapping is that, for types int and bool, we are being consistent with the automatic translation performed by Jessie and Why (recall that CAOVerif translates these types as native Jessie types, and that these frameworks already provide a translation path to Coq). Since all the other CAO types are translated by CAOVerif to logic type declarations in Jessie, which are translated opaquely into Coq definitions by the Jessie and Why tools, this allows us to use our Coq library to support interactive Coq proofs of verification conditions. In other words, as an additional result of this formalization effort, the resulting Coq libraries yield a formalization of the semantics of CAO types that can be used to interactively discharge complex proof obligations that may fall outside of the reach of the axiomatic models included in CAOVerif.

We also remark that our mapping for type mod[τ <X>/p(X)] allows us to validate our axiomatic model for the algebraic operations over extension fields, as this is limited to simple cancellation rules that apply to all finite fields. We are not able to formally verify the axioms that deal with the representation of literals of such types (e.g., stating that the coefficients of the representation of the zero value are all zero). This is the only part of our axiomatic model that is left out of our proof, and this is justified by the lack of support in the adopted version of SSReflect for a formalization of extension field representations.

[5]SSReflect is an extension to the Coq system that includes the formalisation of a large set of mathematical components (`http://www.msr-inria.inria.fr/Projects/math-components`).

[6]In order to use Coq type Z as a type parameter for our SSReflect formalization of container types we had to lift it to an instance of ssreflect's integral domain structure IdomainType.

Formalizing the semantics of CAO operations. The Coq theories we have created for each CAO type include a Coq function for each CAO native operation and Jessie logic function that is explicitly or implicitly axiomatized in CAOVerif. In order to validate such axioms, we first needed to formalize the correct semantics for such operations, by including appropriate definitions of the corresponding Coq functions. Given that the semantics of most CAO types directly matches the mathematical data types included in SSReflect, defining the majority of these Coq functions was simply a matter of identifying the correct SSReflect operation. For example, the logic function that captures addition over mod[n] is simply defined as addition over $'\mathsf{I}_n$. However, for some exceptional cases, the definitions that capture the correct semantics of the operations are slightly more elaborate. As an example, we focus on vector operations.

Recall that we have defined the interpretation domain for vectors as tuples of the appropriate size and type. This means that an operation such as concatenation can be formalized directly by using the SSReflect function cat_tuple. However, our axiomatic models for vectors rely heavily on the blit_vector and shift_vector functions, whose semantics are oblivious to the concrete size of vectors. We therefore captured their semantics in SSReflect sequences, as follows:

```
Definition logic_vector := seq T.

Fixpoint takeD n s : seq T :=
if n is n'.+1
then if s is x::xs then x::takeD n' xs
     else vector_default::takeD n' [::]
else [::].

Definition vector_blit (src dst : logic_vector)
                       (ofs len : nat) : logic_vector :=
        takeD ofs dst ++ takeD len src ++ drop (ofs+len) dst.

Definition vector_shift (v : logic_vector)
                        (i : nat) : logic_vector := drop i v.
```

Here drop and the concatenation ++ operations are included in the seq SSReflect library: drop n s retrieves s after removing its first n items and s_1 ++ s_2 computes the sequence resulting from the concatenation of the sequences s_1 and s_2. Also we define takeD n s to retrieve the sequence of the first n elements of the sequence s, but which appends copies of a distinguished default element to the end of s when its size is less than n.

Proving the validity of the axiomatic models. The entire effort required to formalize and validate the almost 100 lemmas associated with the axiomatic models used in CAOVerif resulted in a total of over 1300 lines of Coq code. Although some time was needed to climb the learning curve of the SSReflect extension, we have found that its context-management tactics and small-scale reflection facilities greatly facilitated our proofs. To illustrate the general flavor of the lemmas we have proven, we return to the vector concatenation example that we have introduced earlier in this section. Establishing the soundness of our model for concatenation involves two steps: 1) showing that our translation of the operation using the blit function is sound; and 2) showing that our axiomatization of the blit function is itself sound. In Coq, these two steps appear as the following lemmas.

```
Lemma vector_get_blit_eq :
  (forall (src:logic_vector), (forall (dst:logic_vector),
    (forall (ofs:nat), (forall (len:nat), (forall (i:nat),
       i >= ofs /\ i < (ofs + len) ->
         vector_get (vector_blit src dst ofs len) i =
                    vector_get src (i - ofs)))))).

Lemma vector_get_blit_neq :
  (forall (src:logic_vector), (forall (dst:logic_vector),
    (forall (ofs:nat), (forall (len:nat),
      (forall (i:nat), (i < ofs) \/ (i >= (ofs + len)) ->
         vector_get (vector_blit src dst ofs len) i =
                   vector_get dst i))))).

Lemma vector_concat_translation :
forall (n1 n2:nat) (u1: n1. tuple T) (u2: n2. tuple T),
 cat_tuple u1 u2 = vector_of_logic_vector (vector_blit u2 u1 n1 n2).
```

The first two lemmas correspond to the axioms that were actually used in our axiomatic model. Hence, they state that our axiomatization of blit using the get and set operations over vectors is sound. The third lemma states the soundness of our translation of the concatenation operation using blit: when seen as a (n1+n2).-tuple,[7] it coincides with the semantics of vector concatenation.

6. Case study: elliptic-curve scalar multiplication in NaCl

We conclude the presentation of CAOVerif by presenting a case study extracted from the CAO implementation of a core component in the open-source NaCl cryptographic library [15]. This component is responsible for carrying out the high-speed elliptic-curve computations required to perform a Diffie-Hellman secret key agreement protocol. At the high-level, given an elliptic curve point (p in the code, and typically a public key) and a scalar (n in the code, and typically a secret key), this component essentially calculates the result of repeatedly adding the given point to itself, where the number of additions is given by the integer value of the scalar. Here, addition should be understood as the group operation defined over the set of points of the particular elliptic curve implemented in NaCl.

The CAO source code for this component is presented in AppendixA and corresponds to a direct transcription of the NaCl specification. The functionality offered by this source code can be summarized as follows.

The entry point into the component is the crypto_scalarmult function, which takes as input two 32-byte arrays. This function then recovers the representation of the elliptic curve point using the unpack function, and also the secret key as a bit string using the clampC function. Function curve25519 is then called to actually perform the elliptic curve computations. This function implements

[7] vector_of_logic_vector is essentially the takeD function mentioned above. Technically, it also attaches a proof that its return value is a sequence of the specified size.

an exponentiation algorithm over a representation of the curve proposed by Montgomery [15]. The exponentiation algorithm in function curve25519 uses as sub-routines the actual curve addition and doubling (adding a point to itself) operations implemented by functions addMont and doubleMont, respectively. These functions operate over a representation of curve points that stores two coordinates x and z, which is captured by the structured type MontRep.

Before presenting our verification results for this case study we first present a small example of the output of our translation into the Jessie input language. This corresponds to function crypto_scalarmult.

```
vector_bits jc_crypto_scalarmult(vector_bits jc_n_input,
                                 vector_bits jc_p_input)
{
   var vector_bits jc_n = jc_n_input;
   var vector_bits jc_p = jc_p_input;
   var mod_25519 jc_pm = mod_25519_of_integer(
                          integer_of_bits(jc_unpack(jc_p)));
   var bits jc_nc = jc_clampC(jc_n);
   return jc_pack(bits_of_integer(integer_of_mod_25519(
                          jc_curve25519(jc_nc, jc_pm))))
}
```

Safety verification. Passing the CAO code in AppendixA to CAOVerif without any annotations gives rise to 309 automatically generated safety proof obligations, most of them arising from accesses to vectors and bit strings. Of these, only 4 proof obligations are not automatically proven by Alt-Ergo, all of them corresponding to function curve25519:

- One VC stating that index i in the bit string access at line 49 is within bounds.
- Two VCs that aim to guarantee loop termination (these are inserted automatically by the Jessie back-end).
- One VC stating that the division in line 63 is safe: CAOVerif can determine that the divisor is not zero because of the test condition in the if statement, but it cannot establish that the divisor is coprime to the modulus $2^{255} - 19$.

The loop annotations in lines 46-47 and the lemma establishing as an hypothesis that $2^{255} - 19$ is a prime number[8] in line 10 are enough to enable CAOVerif to automatically discharge all proof obligations.

Functional correctness verification. To illustrate how CAOVerif can be used to address arbitrary verification goals we introduce a simple example aiming to establish the correctness of function clampC. This function is informally described in the NaCl specification as follows: "*ClampC maps* $(a_0, a_1, ..., a_{30}, a_{31})$ *to* $(a_0 - (a_0 \mod 8), a_1, ..., a_{30}, 64 + (a_{31} \mod 64))$. *In other words, ClampC clears bits* $(7, 0, \ldots, 0, 0, 128)$ *and sets bit* $(0, 0 \ldots, 0, 0, 64)$."

[8]Of course this lemma appears as a non-verified proof obligation at the end of the verification run and one can only hope to verify it interactively.

The bits to be cleared and the bits to be set are specified by the one-bits of the provided values, seen as 8-bit words (bit ordering conventions are specified globally elsewhere in the NaCl specification and we omit them here). The postcondition for clampC in lines 79-83 captures this specification by directly referring to the relation between input and output bits, as this is both clearer and convenient for the proof. Observe that indeed mapping a_0 to $a_0 - (a_0 \mod 8)$ clears the first 3 bits of a_0, and mapping a_{31} to $64 + (a_{31} \mod 64)$ clears the 8th bit and sets the 7th bit of a_{31}.

In order to verify that the clampC function indeed satisfies this specification, we first needed to annotate function unpack with a postcondition (lines 66-68), as this is used by clampC to compute its final result. We also added a set of assertions to guide automatic provers into intermediate verification results that allow them to automatically discharge parts of the postcondition for clampC.

With the annotations included in AppendixA, the CAOVerif is able to discharge all but 1 proof obligation automatically (at least within reasonable time): the postcondition for function unpack. This is due to the large number of nested logic function applications resulting from the translation of the concatenation operations: concatenation is translated in Jessie as a blit_vector operation, hence for 31 concatenations we will have 31 nested blit_vector operations.

We conclude this section with a short description of how we validated this proof obligation using Coq. The postcondition for function unpack expresses that, for $0 \leq i < 31$ and $0 \leq j < 7$, the result of accessing the j-th bit in the i-th bit string in the input vector n is the same as that of accessing the $8i+j$-th position in the concatenated bit string result returned by the function. A simple proof strategy is to exhaustively traverse all the relevant values of i and j and establishing that equality indeed holds. We adopted this strategy, but rather than manually expanding all 256 proof iterations, we developed a simple Coq tactic that implements it based on the following simple lemma.

$$\forall a, b.\ b \leq a \ \vee\ P(a) \wedge (\forall i.\ a+1 \leq i < b \implies P(i)) \implies (\forall i.\, a \leq i < b \implies P(i))$$

Here, P is instantiated with the property we want to prove (in[i][j]=result[i*8+j]), parameterized by the values of i; and a and b are instantiated with the lower and upper bounds of i, respectively.

7. Related work

Formal verification of cryptographic algorithms. Security proofs of high-level cryptographic protocols using formal methods are primarily carried out using symbolic techniques based on the Dolev-Yao model [22], given the potential for automated analysis inherent to this approach. However, it is not always the case that results obtained in such idealized models apply to real systems using concrete cryptographic algorithms. In the last years, significant effort has been put into bridging this gap, either by pursuing the so-called computational soundness results for symbolic proof systems [23], or by developing techniques and tools that enable the formalization and verification of (lower-level) security proofs directly in the computational setting. In this latter category, CertiCrypt [24, 25] is a toolset consisting of two main components, both allowing the formalization of game-based security proofs, but differing in their degree of automation, flexibility and formal guarantees. CertiCrypt, is fully formalized in the Coq proof assistant; its verification methods are implemented in Coq and proved correct w.r.t. program semantics. EasyCrypt, provides more automation by relying on automated theorem provers to discharge verification conditions.

Our work differs from the above in the fundamental aspect that our aim is not to reason about the theoretical security of cryptographic algorithms, but rather about the security properties of *implementations* of those algorithms. We see functional correctness as a necessary (albeit most often not sufficient) condition that permits relating implementation security with theoretical security properties verified using tools such as those described above. In this sense, our view is similar to that in [26], where the authors describe the (hand-crafted) formal verification in Coq of an implementation of the Blum-Blum-Shub [27] (BBS) pseudo-random number generator. The authors integrate the theoretical security analysis framework of [28] with the assembly verification framework of [29], and are able to obtain guarantees of theoretical security for a concrete assembly implementation of the BBS algorithm.

Recent years have seen significant progress in formal verification techniques targeting high-level cryptographic protocols such as TLS [30]. Notable efforts include the automatic inference and security analysis of abstract cryptographic protocol specifications from high-level language implementations [31]; and the automatic synthesis of protocol implementations from abstract protocol descriptions subject to prior verification [32]. We have not yet integrated the program verification capabilities of CAOVerif with a formal verification framework of theoretical security properties. However, the characteristics of CAO place it at an intermediate level of abstraction between assembly and the functional languages typically adopted when targeting high-level protocols [33], which makes CAO an ideal candidate to serve as the implementation language for tools such as CertiCrypt and EasyCrypt, and this is an interesting direction for future work.

There have also been significant advances in the use of domain-specific programming languages (DSLs) for low-level cryptographic implementation. Cryptol [34] is perhaps the DSL that more closely matches the target application area of CAO. Cryptol was developed for the specification and implementation of cryptographic algorithms. It is a functional DSL without global state or side-effects, which was developed with the main purpose of producing formally verified hardware implementations of symmetric cryptographic primitives, such as block ciphers and hash functions. CAO is an imperative language that targets a wider application domain, although also restricted to cryptography. Indeed, the CAO language features have been designed to permit expressing not only symmetric, but also asymmetric cryptographic primitives, in a natural way. Cryptol is supported by a wide range of formal verification tools, which are intended for industrial use and therefore target specific formal verification use cases [35]. CAOVerif aims to enable a much wider range of formal verification use cases and is available under an open-source licence.

Related verification tools. We now revise general purpose program verification tools that are closely related to the one described in this paper. First we note that the verification infrastructure introduced in the Jessie plug-in was already used in the development of other verification tools, in particular Krakatoa [20], a tool for the verification of Java code.

Boogie [36] is a verification condition generator similar in spirit to Why. The input languages to Boogie and Why are both languages with imperative features and first-order assertions, and in both cases verification condition generation is based on a 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# [37]. Boogie performs loop-invariant inference using abstract interpretation and then generates the verification conditions for Simplify or Z3. VCC [38], a tool for low-level concurrent C programs, also has Boogie at its core.

Esc/Java [39] is another deductive verification tool for Java programs whose annotation language is a subset of JML [8], based on an earlier checker for the Modula-3 language. This tool relies on loop unrolling and includes optimisations to avoid an exponential blow-up in weakest-precondition computation. It looks for potential run-time errors in annotated Java programs, but does not model arithmetic overflow. Jack (Java Applet Correctness Kit) [40] is a static verification tool for JML-annotated programs. It provides support for annotation generation and interactive verification of functional specifications, as well as for automatic verification of common security policies and byte-code programs.

One of the oldest and most successful toolsets at the industrial level is SPARK [41]. SPARK is both a programming language (in fact a heavily restricted subset of Ada) targeted at the development of safety-critical software, and a collection of tools for establishing properties of the software. The SPARK toolset is a comercial product developed by Altan Praxis, supporting both automatic and interactive proof. Lately, independent tools have also been proposed for the verification of SPARK code. One such tool is HOL-SPARK [42], which allows proofs to be conducted in the Isabelle [43] interactive proof assistant. Interestingly, the major case study presented in [42] is on the verification of a big number library and an implementation of RSA based on this library, which the authors claim to have taken them three weeks to accomplish.

8. Conclusions

We have presented a model in first-order logic of certain mathematical objects that have specific interest for cryptography, and a concrete approach to using this model for the verification of CAO programs. We believe that the proposed model may be of independent interest and can be of use in other areas, considering that it has been designed to maximise the degree of automation that can be achieved when feeding proof obligations (related to these mathematical abstractions) to general Satisfiability Modulo Theories (SMT) solvers. Given the intricacy of the models, and the complex semantics of the CAO data types, we have formally proven the soundness of our axiomatic models in Coq.

[1] Barbosa, Manuel (editor), CACE Deliverable 5.2: Machine assisted verification and certification tools (June 2010).

[2] T. Ball, S. K. Rajamani, The slam project: debugging system software via static analysis, in: POPL '02: ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 2002, pp. 1–3.

[3] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 2002, pp. 58–70.

[4] R. Jhala, R. Majumdar, Software model checking, ACM Comput. Surv. 41 (4).

[5] J.-C. Filliâtre, C. Marché, The Why/Krakatoa/Caduceus platform for deductive program verification, in: W. Damm, H. Hermanns (Eds.), CAV, Vol. 4590 of LNCS, Springer, 2007, pp. 173–177.

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

[7] M. Barnett, K. Rustan, M. Leino, W. Schulte, The Spec# programming system: An overview, in: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Springer, 2004, pp. 49–69.

[8] 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), ACM, New York, NY, USA, 2000, pp. 105–106.

[9] C. Marché, Y. Moy, Jessie Plugin Tutorial, INRIA (2010).

[10] The Coq Development Team, The Coq Proof Assistant Reference Manual – Version V8.2, `http://coq.inria.fr/refman/` (April 1, 2011).

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

[12] S. Conchon, E. Contejean, F. Bobot, M. Iguernelala, S. Lescuyer, A. Mebsout, Alt-Ergo : an automatic theorem prover dedicated to program verification (`http://alt-ergo.lri.fr`) (April 1, 2011).

[13] L. de Moura, N. Bjorner, Z3: An efficient SMT solver, in: C. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Vol. 4963 of Lecture Notes in Computer Science, Springer Berlin - Heidelberg, 2008, pp. 337–340, 10.1007 978-3-540-78800-3-24.

[14] 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), Vol. 5825 of LNCS, Springer, 2009, pp. 37–52.

[15] D. J. Bernstein, Cryptography in NaCl, `http://nacl.cr.yp.to` (2009).

[16] M. Barbosa, J. S. Pinto, J.-C. Filliâtre, B. Vieira, A deductive verification platform for cryptographic software, in: Proceedings of the Fourth Intl. Workshop on Foundations and Techniques for Open Source Software Certification (Opencert'10), Vol. 33 of ECEASST, 2010.

[17] C. D1.1, Detailed CAO and qhasm language specifications, Tech. Rep. Deliverable D1.1, CACE Project (2009).

[18] M. Barbosa, A. Moss, D. Page, N. F. Rodrigues, P. F. Silva, Type checking cryptography implementations (full version), Tech. Rep. DI-CCTC-11-01, CCTC, Univ. Minho (2011).

[19] M. Barbosa, A. Moss, D. Page, N. F. Rodrigues, P. F. Silva, Type checking cryptography implementations, in: Proceedings of the 4th IPM international conference on Fundamentals of Software Engineering, FSEN'11, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 316–334.

[20] C. Marché, C. Paulin-Mohring, X. Urbain, The Krakatoa tool for certication of Java/JavaCard programs annotated in JML, Journal of Logic and Algebraic Programming (2004) 89–106.

[21] J.-C. Filliâtre, C. Marché, Multi-prover verification of C programs, in: J. Davies, W. Schulte, M. Barnett (Eds.), ICFEM, Vol. 3308 of LNCS, Springer, 2004, pp. 15–29.

[22] D. Dolev, A. C.-C. Yao, On the security of public key protocols, IEEE Transactions on Information Theory 29 (2) (1983) 198–207.

[23] M. Abadi, P. Rogaway, Reconciling two views of cryptography (the computational soundness of formal encryption), J. Cryptology 20 (3) (2007) 395.

[24] G. Barthe, B. Grégoire, S. Zanella Béguelin, Formal certification of code-based cryptographic proofs, in: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '09, ACM, New York, NY, USA, 2009, pp. 90–101.

[25] G. Barthe, B. Grégoire, S. Heraud, S. Zanella Béguelin, Computer-aided security proofs for the working cryptographer, in: CRYPTO 2011, Vol. 6841 of LNCS, Springer, Heidelberg, 2011, pp. 71–90.

[26] R. Affeldt, D. Nowak, K. Yamada, Certifying assembly with formal security proofs: The case of BBS, Sci. Comput. Program. 77 (10-11) (2012) 1058–1074.

[27] L. Blum, M. Blum, M. Shub, A simple unpredictable pseudo random number generator, SIAM J. Comput. 15 (2) (1986) 364–383.

[28] D. Nowak, A framework for game-based security proofs, in: ICICS, Vol. 4861 of Lecture Notes in Computer Science, Springer, 2008, pp. 319–333.

[29] R. Affeldt, N. Marti, An approach to formal verification of arithmetic functions in assembly, in: Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues, ASIAN'06, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 346–360.

[30] T. Dierks, E. Rescorla, The transport layer security (tls) protocol, in: IETF RFC 4346, 2006.

[31] M. Aizatulin, A. D. Gordon, J. Jürjens, Extracting and verifying cryptographic models from C protocol code by symbolic execution, in: Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, ACM, New York, NY, USA, 2011, pp. 331–340.

[32] C. Fournet, G. Le Guernic, T. Rezk, A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms, in: Proceedings of the 16th ACM conference on Computer and communications security, CCS '09, ACM, New York, NY, USA, 2009, pp. 432–441.

[33] K. Bhargavan, C. Fournet, R. Corin, E. Zalinescu, Cryptographically verified implementations for tls, in: Proceedings of the 15th ACM conference on Computer and communications security, CCS '08, ACM, New York, NY, USA, 2008, pp. 459–468.

[34] J. Lewis, Cryptol: specification, implementation and verification of high-grade cryptographic applications, in: FMSE '07, ACM, 2007, p. 41.

[35] L. Erkök, J. Matthews, High assurance programming in cryptol, in: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies, CSIIRW '09, ACM, New York, NY, USA, 2009, pp. 60:1–60:2.

[36] 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), Vol. 4111 of LNCS, Springer-Verlag, 2006, pp. 364–387.

[37] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, H. Venter, Specification and verification: the Spec# experience, Commun. ACM 54 (6) (2011) 81–91.

[38] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, Vcc: A practical system for verifying concurrent C, in: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Eds.), TPHOLs, Vol. 5674 of Lecture Notes in Computer Science, Springer, 2009, pp. 23–42.

[39] 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), ACM, 2002, pp. 234–245.

[40] 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: F. S. de Boer, M. M. Bonsangue, S. Graf, W. P. de Roever (Eds.), FMCO, Vol. 4709 of Lecture Notes in Computer Science, Springer, 2006, pp. 152–174.

[41] J. Barnes, High Integrity Software: The SPARK Approach to Safety and Security, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[42] S. Berghofer, Verification of dependable software using spark and isabelle, in: J. Brauer, M. Roveri, H. Tews (Eds.), SSV, Vol. 24 of OASICS, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, pp. 15–31.

[43] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer, 2002.

AppendixA. CAO implementation of crypto_scalar_mult

```
typedef byte     := unsigned bits[8];
typedef unpacked := unsigned bits[256];
typedef packed   := vector[32] of unsigned bits[8];
typedef skey     := unsigned bits[255];
typedef Fp       := mod[2**255-19];

/* Curve points in Montgomery representation */
typedef MontRep  := struct [ def x : Fp; def z : Fp; ];

/*@ lemma is_prime_Fp: is_prime(2**255-19); */

/* Constant global curve parameter */
def a2 : Fp := [486662];
/*@ global invariant constant_a2: a2==[486662] */

/* Curve point addition */
def addMont(Q,Qpr,QmQpr : MontRep) : MontRep {
    def Q3 : MontRep;

    Q3.x := [4] * (Q.x * Qpr.x - Q.z*Qpr.z)**2 * QmQpr.z;
    Q3.z := [4] * (Q.x * Qpr.z - Q.z*Qpr.x)**2 * QmQpr.x;

    return Q3;
}

/* Curve point addition */
def doubleMont(Q : MontRep) : MontRep {
    def Q2 : MontRep;

    Q2.x := (Q.x**2 - Q.z**2)**2;
    Q2.z := [4]*Q.x*Q.z*(Q.x**2+a2*Q.x*Q.z+Q.z**2);

    return Q2;
}

/* Curve point scalar multiplication */
def curve25519(n : skey, base : Fp) : Fp {
    def i : int := 253;
    def mth, mp1th, one : MontRep;

    one.x := base;
    one.z := [1];
    mth := one;
    mp1th := doubleMont(one);

    /*@ invariant -1<=i<=253
        variant i */
    while(i>=0) {
        if (n[i] == 1) {
            mth := addMont(mth,mp1th,one);
            mp1th := doubleMont(mp1th);
        }
        else {
            mp1th := addMont(mth,mp1th,one);
            mth := doubleMont(mth);
        }
        i := i-1;
    }
    if (mth.z == [0]) {
        return [0];
    }
    else {
        return (mth.x/mth.z);
    }
}
```

```
/* Unpacking a byte array */
/*@ ensures (forall i,j,k:int;
        (0<=i<32 && 0<=j<8 && k==i*8+j) ==> in[i][j]==result[k]) */
def unpack(in : packed) : unpacked {
    return (in[ 0] @ in[ 1] @ in[ 2] @ in[ 3] @ in[ 4] @ in[ 5] @
            in[ 6] @ in[ 7] @ in[ 8] @ in[ 9] @ in[10] @ in[11] @
            in[12] @ in[13] @ in[14] @ in[15] @ in[16] @ in[17] @
            in[18] @ in[19] @ in[20] @ in[21] @ in[22] @ in[23] @
            in[24] @ in[25] @ in[26] @ in[27] @ in[28] @ in[29] @
            in[30] @ in[31]);
}

/* Reconstructing the secret key */
/*@ ensures result[0]== 0b0 && result[1  ]==0b0 &&
            result[2]== 0b0 && result[254] == 0b1
    ensures (forall i,j:int;
            0<=i<32 && 0<=j<8 && 2<i*8+j<254 ==>
            n[i][j]==result[i*8+j])
 */
def clampC(n : packed) : skey {
   def key: skey;
   def pack : unpacked;

   pack := unpack(n);

   pack[0..2] := 0b000;
   /*@ assert pack[0]==0b0 */
   /*@ assert pack[1]==0b0 */
   /*@ assert pack[2]==0b0 */

   pack[254..255]  := 0b01;
   /*@ assert pack[254]==0b1 */
   /*@ assert pack[255]==0b0 */

   /*@ assert (forall i,j:int;
         0<=i<32 && 0<=j<8 && 2<i*8+j<254 ==> n[i][j] == pack[i*8+j])*/

    key := pack[0..254];
    /*@ assert (forall k:int; 0<=k<=254 ==> key[k] == pack[k])*/
    /*@ assert (forall i,j:int;
         0<=i<32 && 0<=j<8 && 2<i*8+j<254 ==> n[i][j] == key[i*8+j])*/
    return key;
}

/* Packing a byte array */
def pack(in : unpacked) : packed {
    def out : packed;

    seq i := 0 to 31 {
        out[i]:=in[8*i..8*i+7];
    }

    return out;
}

/* Entry point */
def crypto_scalarmult(n,p : packed) : packed {
    def pm : Fp := (Fp)unpack(p);
    def nc : skey := clampC(n);

    return(pack((unpacked)(int)curve25519(nc,pm)));
}
```