Type Checking Cryptography Implementations

Manuel Barbosa[1] Andrew Moss[2] Dan Page[3]
Nuno F. Rodrigues[1,4] Paulo F. Silva[1]

[1] Departamento de Informática, Universidade do Minho
[2] School of Computing, Blekinge Institute of Technology
[3] Department of Computer Science, University of Bristol
[4] DIGARC, Instituto Politécnico do Cávado e do Ave

Abstract. Cryptographic software development is a challenging field: high performance must be achieved, while ensuring correctness and compliance with low-level security policies. CAO is a domain specific language designed to assist development of cryptographic software. An important feature of this language is the design of a novel type system introducing native types such as predefined sized vectors, matrices and bit strings, residue classes modulo an integer, finite fields and finite field extensions, allowing for extensive static validation of source code. We present the formalisation, validation and implementation of this type system.

1 Introduction

The development of cryptographic software is clearly distinct from other areas of software engineering. The design and implementation of cryptographic software draws on skills from mathematics, computer science and electrical engineering. Also, since security is difficult to sell as a feature in software products, cryptography needs to be as close to invisible as possible in terms of computational and communication load. As a result, cryptographic software must be optimised aggressively, without altering the security semantics. Finally, cryptographic software is implemented on a very wide range of devices, from embedded processors with very limited computational power and memory, to high-end servers, which demand high-performance and low-latency. Therefore, the implementation of cryptographic kernels imposes a specific set of challenges that do not apply to other system components. For example, direct implementation in assembly language is common, not only to guarantee a more efficient implementation, but also to ensure that low-level security policies are satisfied by the machine code.

The CAO language. The CAO language aims to change this state of affairs, allowing natural description of cryptographic software implementations, which can be analysed by a compiler that performs security-aware analysis, transformation and optimisation. The driving principle behind the design of CAO is that the language should support cryptographic concepts as first-class language features. Unlike the languages used in mathematical software packages such as Magma or Maple, which allow the description of high-level mathematical constructions in their full generality, CAO is restricted to enabling the implementation of cryptographic components such as block ciphers, hash functions and sequences of finite field arithmetic for Elliptic Curve Cryptography (ECC).

CAO preserves some higher-level features to be familiar to an imperative programmer, whilst focusing on the implementation aspects that are most critical for security and efficiency. The memory model of CAO is, by design, extremely simple to prevent memory management errors (there is no dynamic memory allocation and it has call-by-value semantics). Furthermore, the language does not support any input/output constructions, as it is targeted at implementing the core components in cryptographic libraries. In fact, a typical CAO program comprises only the definition of a global state and a set of functions that permit performing cryptographic operations over that state. Conversely, the native types and operators in the language are highly expressive and tuned to the specific domain of cryptography. In short, the design of CAO allowed trading off the generality of a language such as C or Java, for a richer type system that permits expressing cryptographic software implementations in a more natural way.

CAO introduces as first-class features pure incarnations of mathematical types commonly used in cryptography (arbitrary precision integers, ring of residue classes modulo an integer, finite field of residue classes modulo a prime, finite field extensions and matrices of these mathematical types) and also bit strings of known finite size. A more expressive type system would be expected from any domain-specific language. However, in the case of CAO, the design of the type system was taken a step further in order not only to allow an elegant formalisation of the type checking rules, but also to allow the efficient implementation of a type checking system that performs extensive preliminary validation of the code, and extracts a very rich body of information from it. This fact makes the CAO type checker a critical building block in the implementation of compilation and formal verification tools supporting the language.

Contributions. This paper presents the formalisation, validation and implementation of the CAO type system. Our main contribution is to show that the trade-offs in language features that were introduced in the design of CAO – specifically for cryptographic software implementation – enabled us to tame the complexity of formalising and validating a surprisingly powerful type system. We also show, resorting to practical examples, how this type system enforces strong typing rules and how these rules detect several common run-time errors. To support this claim, we outline our proof of soundness of the CAO type system.

More in detail, we describe a formalisation of the CAO type system and the corresponding implementation of a type checker[5] as a front-end of the CAO tool chain. One of the main achievements of our system is the enforcement of strong typing rules that are aware of type parameters in the data types of the language. The type checking rules permit determining concrete values for these parameters and, furthermore, resolving the consistency of these parameters inside CAO programs. Concretely, the CAO type system explicitly includes as type

[5] An implementation of a CAO interpreter (including the type system and semantics) is available via `http://www.cace-project.eu`.

parameters the sizes of containers such as vectors, matrices and bit strings. In other words, CAO is dependently typed. Furthermore, typing of complex operations over these containers, including concatenation and extensional assignment, statically checks the compatibility of these parameters.

More interestingly, we are able to handle parameters in mathematical types in a similar way. Our type system maintains information for the concrete values of integer moduli and polynomial moduli, so that it is possible to validate the consistency of complex mathematical expressions, including group and finite field operations, the conversion between a finite field element and its polynomial representation, and other type conversions. Finally, the CAO type system also deals with language usability issues that include implicit (automatic) type conversions between bit strings and the integer value that they represent, and also between values within the same finite field extension hierarchy.

Paper organisation. In Section 2 we expand on the relevant features of CAO. We then build some intuition for the subsequent formal presentation of the type system by introducing real-world examples of CAO code in Section 3. In Section 4 we present the CAO type system, including a detailed example of its operation. In Section 5 we describe our implementation. We conclude with a discussion of soundness and related work in Sections 6 and 7.

2 A closer look at CAO

Real world examples of the most relevant CAO language features are presented in Section 3. We now provide an intuitive description of the CAO type system.

Bit strings. The bits type represents a string of n bits (labelled $0 \ldots n{-}1$, where the 0-th is the least-significant bit). This should not be seen as the "bit vector" type, as the get operator a[i] actually returns type bits[1]. The distinction between ubits and sbits concerns only the conversion convention to the integer type, which can be unsigned or two's complement respectively. The bits type is equipped with a set of C-like bit-wise operators, including the usual Boolean, shift and rotate operators, which are closed over the bit-length. The range selection/assignment (or slicing) operator (..), combined with the concatenation operator @ can be used to (de)construct bit strings of different sizes using a very concise syntax. For example, the following is a valid CAO statement over bit strings:

```
a[3..8] := b[0..2] @ c[2..4];
```

Integers and the mod type. Operations modulo some prime or composite integer are used extensively in cryptography [6]; for example, the ring[6] $\mathbb{Z}_n$ underlies the pervasively used RSA function [4], and the finite field[7] $\mathbb{F}_p$ is widely

[6] The ring of residue classes modulo an integer n can be seen as the set of numbers in the range 0 to n-1 with addition and multiplication modulo n.

[7] The ring of residue classes modulo an integer p is actually a field when p is prime: all non-zero elements have a multiplicative inverse.

used in ECC. Therefore, CAO includes not only arbitrary precision integers as a native type (int), but also a mod[n] type. For example, the mod[7] type is an instance of mod with modulus 7. In this case the modulus is prime, and hence inhabitants of this type are actually elements of a finite field. More generally, the modulus can be prime or composite, provided it is fixed at compile-time. Algebraic operations over the mod type are closed over the modulus parameter.

Internal representation and Casts. The internal representation of mathematical types is deliberately undefined. The CAO semantics ensures that arithmetic with such values is valid, but makes no guarantee about (and hence disallows access to) their physical representation. Nevertheless, the CAO type system includes the necessary functionality to access the conceptually natural representation of algebraic types, by supporting appropriate cast operators. For example, to obtain the representation of a finite field element in mod[p] as an integer value of the appropriate range, one simply casts it into the int type. To obtain the representation of an arbitrary precision integer, one can cast it into a bit string of a predetermined size, and so on. Hence, compared to C, a CAO cast is more explicitly a conversion. Aside from this nuance, the syntax of casts is similar to C: one specifies the target type in parenthesis, e.g. y := (int) x.

General moduli. An alternative form of the mod type allows defining finite field extensions, as shown below:

```
typedef a := mod[ 2 ];
typedef b := mod[ a<X> / X**8 + X**4 + X**3 + X + 1 ];
```

The type synonym a represents a mod type whose modulus is 2; this is simply the field $\mathbb{F}_2$. This is used as the base type for a second type synonym b which represents the field $\mathbb{F}_{2^8}$. In addition to the base type one also specifies an indeterminate symbol (in this case X), and an irreducible polynomial in the ring of polynomials with coefficients in the base type (in this case $P(X) = X^8+X^4+X^3+X+1$). Intuitively, this declaration defines an implementation of the field based on the referred polynomial ring, with arithmetic defined via standard polynomial algebra with reductions modulo $P(X)$. To access the coefficients in this representation, one can cast the value into a vector of elements in the base type.

Matrices. The matrix type represents a 2-dimensional algebraic matrix over which one can perform addition and multiplication. For this reason, there are some restrictions on what the base type can be. The matrix type also has an undefined representation; its size must be fixed at compile-time, but the ordering of elements in memory (e.g. row-major or column-major order) is a choice that can be made by the compiler. The matrix type also supports get and range selection/assignment operations that permit easily (de)constructing matrices of different sizes.

Vectors. The vector type represents a 1-dimensional generic container of elements of homogeneous type, where each element is referred to by a single index in the range $0 \ldots n-1$, offering selection/assignment, concatenation and rotate operations similar to the bits type.

3 CAO Type System in Action

In this section we present some examples of CAO code taken from the implementation of the NaCl cryptographic library[8] that illustrate the validation capacity of the type checker over real world examples.

The following program fragment was taken from the implementation of the poly1305 one-time message authentication mechanism [2]. The function receives two vectors ciu and ru of content type byte, which is an alias for type unsigned bits[8], and an integer q. It returns a value of type mod1305, an alias for type mod[2**130-5].

```
def polyStep(ciu:vector[17] of byte, ru:vector[16] of byte, q:int) : mod1305 {
    def r : unsigned bits[16*8]; def ci : unsigned bits[17*8];

    r := ru[0]@ru[1]@ru[2]@ru[3]@ru[4]@ru[5]@ru[6]@ru[7]@ru[8]@ru[9]@ru[10]@
         ru[11]@ru[12]@ru[13]@ru[14]@ru[15];

    ci:= ciu[0]@ciu[1]@ciu[2]@ciu[3]@ciu[4]@ciu[5]@ciu[6]@ciu[7]@ciu[8]@
         ciu[9]@ciu[10]@ciu[11]@ciu[12]@ciu[13]@ciu[14]@ciu[15]@ciu[16];

    return ((mod1305)ci * (mod1305)r**q); }
```

The type system must solve the following problems to type the function body. Firstly, the concatenation of several bit strings must be typed to a single bit string of the appropriate type and size (and fail if these do not match in assignment). Secondly, the type checker must recognise that the cast to type mod1305 requires the expression on the right to be coerced to type int.

The next program fragment is from the NaCl implementation of hsalsa20 [3].

```
seq i := 0 to 3 {
    x[i+1]  :=  from_littleendian( k[i*4..i*4+3]);
    x[i+6]  :=  from_littleendian(in[i*4..i*4+3]);
    x[i+11] :=  from_littleendian( k[i*4+16..i*4+19]); }
...
seq i := 0 to 3 {
    out[i*4..i*4+3]     := to_littleendian(x[5*i]);
    out[i*4+16..i*4+19] := to_littleendian(x[i+6]); }
```

This is a good example of how CAO was fine tuned to provide assistance to the programmer in what, at first sight, might seem like a surprisingly powerful validation procedure. Range selection and assignment operators in bit strings, vectors and matrices may depend on the value of integer expressions, which can only be formed by literals, constants and basic arithmetic operations that can be evaluated at compile-time. This might seem just like a pre-processing step of compilation, were it not for the fact that we are also able to include in these expressions locally defined constants. Our type system is able to validate that all range selections (resp. assignments) result in vectors that are compatible with calls to function from_littleendian (resp. return type of function to_littleendian).

Finally, the following code snippet is extracted from a CAO implementation of AES. It shows how our type system is capable of dealing with the complex mathematical types that arise in cryptographic implementations. In this case we have a matrix multiplication operation mix * s[0..3,i], where the contents of the matrices are elements of a finite field extension GF2N.

[8] http://nacl.cr.yp.to

n : **Num**	Numerals	pg : **Progs**	Programs
x : $\mathbf{Id}_V$	Variable Identifiers	e : **Exp**	Expressions
fp : $\mathbf{Id}_{FP}$	Function and Procedure Identifiers	c : **Stm**	Statements
dv : $\mathbf{Dec}_V$	Variable declarations	l : **Lv**	LValues
dfp : $\mathbf{Dec}_{FP}$	Function and Procedure declarations	pol : **Poly**	Polynomials
ds : $\mathbf{Dec}_S$	Struct declarations	t : **Types**	Types

$$
\begin{aligned}
e &::= n \mid \mathsf{true} \mid \mathsf{false} \mid x \mid -e \mid e_1 \dagger e_2 \mid e.x \mid e_1[e_2] \mid e_1[e_2..e_3] \mid \\
&\quad\; e_1[e_2, e_3] \mid e_1[e_2..e_3, e_4..e_5] \mid\sim e \mid (t)\ e \mid fp(e_1, ..., e_n) \mid !\ e \\
l &::= x \mid l.x \mid l[e] \mid l[e_1..e_2] \mid l[e_1, e_2] \mid l[e_1..e_2, e_3..e_4] \\
c &::= dv \mid l_1, ..., l_i := e_1, ..., e_j \mid c_1; c_2 \mid \mathsf{if}\ (e)\ \{\ c_1\ \} \mid \mathsf{if}\ (e)\ \{\ c_1\ \}\ \mathsf{else}\ \{\ c_2\ \} \mid \\
&\quad\; \mathsf{while}\ (e)\ \{\ c\ \} \mid \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} \mid \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} \mid \\
&\quad\; \mathsf{return}\ e_1, ..., e_n \mid fp(e_1, ..., e_n) \\
dv &::= \mathsf{def}\ x_1, ..., x_n : t_1, ..., t_n \mid \mathsf{def}\ x_1, ..., x_n : t_1, ..., t_n{:=}e_1, ..., e_n \\
ds &::= \mathsf{typedef}\ x := t; \mid \mathsf{typedef}\ x_1 := \mathsf{struct}\ [\ \mathsf{def}\ x_2 : t_1; ...; \mathsf{def}\ x_n : t_n\ ]; \\
dfp &::= \mathsf{def}\ fp\ (x_1 : t_1, ..., x_n : t_n) : rt\ \{\ c\ \} \\
rt &::= \mathsf{void} \mid t_1, \ldots, t_n \\
t &::= x \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{signed\ bits}\ [e] \mid \mathsf{unsigned\ bits}\ [e] \mid \mathsf{mod}\ [e] \mid \mathsf{mod}\ [\ t\ x\ /\ pol\ ] \mid \\
&\quad\; \mathsf{vector}\ [n]\ \mathsf{of}\ t \mid \mathsf{matrix}\ [n_1, n_2]\ \mathsf{of}\ t \\
pg &::= dv; \mid ds \mid dfp \mid pg_1\ pg_2
\end{aligned}
$$

Fig. 1: Formal syntax of CAO

```
typedef GF2  := mod[ 2 ];
typedef GF2N := mod[ GF2<X> / X**8 + X**4 + X**3 + X + 1 ];
typedef S    := matrix[4,4] of GF2N;

def mix : matrix[4,4] of GF2N  :=
  {[X],[X+1],[1],[1],[1],[X],[X+1],[1],[1],[1],[X],[X+1],[X+1],[1],[1],[X]};

def MixColumns( s : S ) : S {
  def r : S;
  seq i := 0 to 3 { r[0..3,i] := mix * s[0..3,i]; }
  return r; }
```

In addition to resolving the matrix size restrictions imposed by the matrix multiplication operation, our type system is able to individually type the finite field literals in the matrix initialisation, and check that these types are compatible with the type of the matrix contents. Note that this implies recognising that a literal of type mod[2] is coercible to GF2N.

4 Formalisation of the CAO Type System

In this Section, we will overview our formalisation of the CAO type system. Since CAO is a relatively large language, only the most interesting features will be covered. A full description of the CAO formalisation can be found in [1].

CAO Syntax. The formal syntax of CAO is presented in Figure 1. To simplify presentation we use † to represent a set of traditional binary operators, namely

$$\dagger \in \{+, -, *, /, \%, **, \&, \hat{}, |, \gg, \ll, @, ==, !=, <, >, <=, >=, ||, \&\&, \hat{}\hat{}\}$$

Most of the binary operators are the same as their C equivalents, although they are overloaded for multiple types. Worth mentioning are the multiplicative exponentiation operator for integers, residue class groups and fields ($**$); the bit-wise conjunction (AND), inclusive- (IOR) and exclusive-disjunction (XOR) operators (&, | and ˆ respectively); the shift operators for bit strings and vectors ($\gg$ and $\ll$); the concatenation operator for bit strings and vectors @; and the boolean logic exclusive-disjunction (XOR) operator (ˆˆ).

Most of the language syntactic entities, and the accompanying syntax rules, are also similar to C. Additional domains have been added to this basic set: some for the sake of a clearer presentation, and others because they are part of CAO's domain specific character for cryptography.

4.1 CAO Type System

Function Classification. The type checker is able to automatically classify CAO functions with respect to their interaction with global variables. The type checking rules classify functions as either of the following three types:

Pure functions Do not depend on global variables in any way and can only call other pure functions. These functions are, not only side-effect free, but also return the same result in every invocation with the same input. This property is often called referential transparency.

Read-only functions Can read values from global variables, but they cannot assign values to them. They can call pure functions and other read-only functions, but not procedures. These functions are side-effect free.

Procedures Can read and assign values from/to global variables. They can call pure functions, read-only functions and other procedures.

For the CAO type checker, the most important distinction is that between procedures and other functions. Procedures are only admitted in restricted contexts, such as simple assignment constructions. This distinction is completely automated in the type-checking rules that associate the following total order of classifiers to CAO constructions: $\mathsf{Pure} < \mathsf{ReadOnly} < \mathsf{Procedure}$

Put simply, the type checking system enforces the following rules: 1) A construction depending only on local variables is classified as Pure; 2) When reading the value of a global variable, the classifier is set to Read-only; 3) When a global variable is used in an assignment target, the classifier is set to Procedure; 4) Expressions and statements procedures are classified with respect to their sub-elements using the *maximum operator* defined over the total order specified above. Note that this classification system is conservative in the sense that, for example, it will fail to correctly classify a function as pure when it reads a global variable but does not use its value.

Environments, type judgements and conventions. We use symbol τ (possibly with subscripts) to represent an arbitrary (fixed) data type. We write $x :: \tau$ to denote that x has type τ. We use two distinct environments in our type rules:

the type environment relation $\mathit{\Gamma}$, which collects all the declarations (e.g. variables, function, procedures) together with their associated types; and the constant environment relation $\mathit{\Delta}$, which records the values associated with integer constants. The $\mathit{\Gamma}$ environment is partitioned into two relations: $\mathit{\Gamma}_G$ for global definitions and $\mathit{\Gamma}_L$ for local definitions. This distinction is important to deal with symbol scoping and visibility when typing, for example function declarations. Whenever this distinction is not important we will just write $\mathit{\Gamma}$ to abbreviate $\mathit{\Gamma}_G, \mathit{\Gamma}_L$. Notation $\mathit{\Gamma}[x :: \tau]$ is used to extend the environment $\mathit{\Gamma}$ with a new variable x of type τ, providing that x is not in the original environment (i.e., $x \notin \mathit{dom}(\mathit{\Gamma})$). Similarly, $\mathit{\Delta}[x := n]$ is used to extend the environment $\mathit{\Delta}$ with a new constant x with value n, also provided that x is not in the domain of environment $\mathit{\Delta}$. Notation $\mathit{\Gamma}(x)$ and $\mathit{\Delta}(x)$ represent, respectively, the type and the integer value associated with identifier x, assuming that x belongs to the domain of the respective environment. Environments are built by order of declaration in source code, implying that recursive declarations are not possible and that function classifiers are already known when the functions are called.

We use symbol $\vdash$ for type judgement of expressions of the form $\mathit{\Gamma}, \mathit{\Delta} \vdash e :: (\tau, \mathsf{c})$, retrieving type τ and functional classifier c associated to an expression. Operator $\Vdash_\beta$ denotes type judgements of statements that may modify the type environment relation: it retrieves not only a typed statement, but also a new type environment relation. Subscript β (seen as a place-holder) in operator $\Vdash_\beta$ represents the return type of the function in which the statement was defined. This information is particularly useful, allowing the type checker to guarantee that the several return statements that may appear in a function are always in accordance with the return type of the corresponding function declaration.

Evaluation of integer expressions. We define a partial function $\phi_{\mathit{\Delta}}$ to deal with type parameters such as vector sizes that must be determined at compile time. This function is used in typing rules to compute the integer value of a given expression e in context $\mathit{\Delta}$. If this value cannot be determined, then typing will fail. This function is defined as follows

$$\begin{array}{cc}
\phi_{\mathit{\Delta}}(n) = n & \phi_{\mathit{\Delta}}(x) = \mathit{\Delta}(x),\ x \in \mathit{dom}\ \mathit{\Delta} \\
\phi_{\mathit{\Delta}}(-e) = -\phi_{\mathit{\Delta}}(e) & \phi_{\mathit{\Delta}}(e_1 \dagger e_2) = \phi_{\mathit{\Delta}}(e_1) \dagger \phi_{\mathit{\Delta}}(e_2) \\
\phi_{\mathit{\Delta}}(e_1 ** e_2) = (\phi_{\mathit{\Delta}}(e_1))^{(\phi_{\mathit{\Delta}}(e_2))} & \phi_{\mathit{\Delta}}(e_1 \,\%\, e_2) = \phi_{\mathit{\Delta}}(e_1) \mod \phi_{\mathit{\Delta}}(e_2)
\end{array}$$

for $\dagger \in \{+, -, *, /\}$. When evaluating integer expressions in typing rules, we write

$$\frac{\ldots \quad \phi_{\mathit{\Delta}}(e) = n \quad \ldots}{\mathit{\Gamma}, \mathit{\Delta} \vdash \ldots} \quad \text{to mean} \quad \frac{\ldots \quad \mathit{\Gamma}, \mathit{\Delta} \vdash e :: (\mathsf{Int}, \mathsf{Pure}) \quad \phi_{\mathit{\Delta}}(e) = n \quad \ldots}{\mathit{\Gamma}, \mathit{\Delta} \vdash \ldots}$$

which implicitly implies that expression e is of integer type.

Data types. In Section 2, types were informally described using CAO syntax for type declarations. Here we will distinguish between a type declaration and the type it refers to in our formalisation. We use upper case to indicate the CAO

Table 1: CAO data types.

Bool	Booleans
Int	Arbitrary precision integers
UBits $[i]$	Unsigned bit strings of length i
SBits $[i]$	Signed bit strings of length i
Mod $[n]$	Rings or fields defined by integer n
Mod $[\tau/pol]$	Extension field defined by τ/pol
Vector $[i]$ of τ	Vectors of i elements of type τ
Matrix $[i,j]$ of α	Matrices of $i \times j$ elements of type $\alpha \in \mathcal{A}$

$$\mathcal{A} = \{\mathsf{Int}, \mathsf{Mod}\ [m], \mathsf{Matrix}\ [i,j]\ \mathsf{of}\ \alpha \ \mid \alpha \in \mathcal{A}\}$$

data types shown in Table 1. An important difference is that the CAO grammar allows any expression as a parameter of a type declaration, while CAO types must have parameters of the correct type and with a fully determined value, e.g., sizes must be integer values. In Table 1, $\mathcal{A}$ denotes the set of algebraic types, which are the only ones that can be used to construct matrices. These are types for which addition, multiplication and symmetric operators are closed. In order to emphasise occurrences where the type must be algebraic, we will use α (possibly with subscripts) instead of τ.

Type translation. To deal with the type parameters informally described in Section 1, we introduce a new judgement that makes the translation between type declaration in the CAO syntax and types used in the type checking process. This judgement, of the form $\Delta \vdash_t t \rightsquigarrow \tau$, depends only on the environment Δ, which can in turn be used to determine the values of expressions that only depend on constants. This accounts for the fact that, during type checking, types must have their parameters fully determined, while type declarations in CAO can depend on arithmetic expressions using constants stored in the environment Δ. Hence the translation judgement uses evaluation function ϕ_Δ to compute parameter expressions in the declaration of bit string, vector and matrix sizes, ensuring that no negative or zero sizes are used. The evaluation function is also used in modular types with integer modulus to determine its value and ensure that it is meaningful (i.e., greater than 1). We present only part of this definition below.

$$\frac{\phi_\Delta(e) = n}{\Delta \vdash_t \mathsf{unsigned\ bits}\ [e] \rightsquigarrow \mathsf{UBits}[n]}\, n \geq 1 \qquad \frac{\phi_\Delta(e) = n}{\Delta \vdash_t \mathsf{mod}\ [e] \rightsquigarrow \mathsf{Mod}[n]}\, n \geq 2$$

$$\frac{\phi_\Delta(e) = n \quad \Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \vdash_t \mathsf{vector}\ [e]\ \mathsf{of}\ t \rightsquigarrow \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau}\, n \geq 1$$

$$\frac{\phi_\Delta(e_1) = n \quad \phi_\Delta(e_2) = m \quad \Delta \vdash_t t \rightsquigarrow \alpha}{\Delta \vdash_t \mathsf{matrix}\ [e_1, e_2]\ \mathsf{of}\ t \rightsquigarrow \mathsf{Matrix}\ [n, m]\ \mathsf{of}\ \alpha} \quad \alpha \in \mathcal{A}, n \geq 1, m \geq 1$$

Type coercions. Type coercions are essentially implicit (typically data preserving) type conversions, whereby the programmer is allowed to use terms of some type whenever another type is expected. In CAO, this mechanism is re-

Table 2: Type coercion relation, $\vdash_{\leq} t_1 \leq t_2$

$\mathbf{t_1}$	$\mathbf{t_2}$	**Condition**
$\mathsf{UBits}[n]$	Int	
$\mathsf{SBits}[n]$	Int	
τ	$\mathsf{Mod}[\tau'/pol]$	$\vdash_{\leq} \tau \leq \tau'$
$\mathsf{Vector}[n]$ of τ_1	$\mathsf{Vector}[n]$ of τ_2	$\vdash_{\leq} \tau_1 \leq \tau_2$
$\mathsf{Matrix}\ [i,j]$ of α_1	$\mathsf{Matrix}\ [i,j]$ of α_2	$\vdash_{\leq} \alpha_1 \leq \alpha_2$ and $\alpha_1, \alpha_2 \in \mathcal{A}$

Table 3: A few cases for the cast relation, $\vdash_c t_1 \Rightarrow t_2$.

$\mathbf{t_1}$	$\mathbf{t_2}$	**Condition**
Int	$\mathsf{Bits}\ [i]$	
Int	$\mathsf{Mod}\ [n]$	
$\mathsf{Vector}\ [i]$ of τ_1	$\mathsf{Mod}\ [\tau_2/pol]$	$\vdash_c \tau_1 \Rightarrow \tau_2$ and $i = degree(pol)$
$\mathsf{Mod}\ [\tau_1/pol]$	$\mathsf{Vector}\ [i]$ of τ_2	$\vdash_c \tau_1 \Rightarrow \tau_2$ and $i = degree(pol)$
$\mathsf{Matrix}\ [1,j]$ of α	$\mathsf{Vector}\ [j]$ of τ	$\vdash_c \alpha \Rightarrow \tau$ and $\alpha \in \mathcal{A}$
$\mathsf{Vector}\ [i]$ of τ	$\mathsf{Matrix}\ [i,1]$ of α	$\vdash_c \tau \Rightarrow \alpha$ and $\alpha \in \mathcal{A}$
$\mathsf{Vector}\ [i]$ of τ_1	$\mathsf{Vector}\ [i]$ of τ_2	$\vdash_c \tau_1 \Rightarrow \tau_2$
$\mathsf{Matrix}\ [i,j]$ of α_1	$\mathsf{Matrix}\ [i,j]$ of α_2	$\vdash_c \alpha_1 \Rightarrow \alpha_2$ and $\alpha_1, \alpha_2 \in \mathcal{A}$

markably useful, for example when dealing with field extensions (cf. the third rule in Table 2), since a field can be seen as a subtype of all its field extensions. In general, when a CAO type τ_1 is coercible to another type τ_2, then the set of values in τ_1 can be seen as a subset of the values in τ_2. For example, all bit-strings of a given size can be coerced to the integer type. We define a coercion relation $\leq$, associated with a new kind of judgement $\vdash_{\leq}$. Coercions are naturally reflexive, and Table 2 summarises the other possible coercions.

Often the arguments of an operation have different types but are coercible to a common type, or one is coercible to the other. In order to capture this situation, we define the $\uparrow$ operator on types, which returns the least upper bound of the types to which its arguments are coercible:

$$\tau_1 \uparrow \tau_2 = \min\{\tau \mid \vdash_{\leq} \tau_1 \leq \tau \text{ and } \vdash_{\leq} \tau_2 \leq \tau\}$$

This requires that the coercion relation $\leq$ is regarded as a partial order on types, thus requiring the reflexivity, transitivity and anti-symmetry properties to hold. As we have seen before, the coercion relation is reflexive; the transitivity and anti-symmetry requirements are also easy to add and well suited to our intuitive notion of coercion. With these properties in place, and for the particular set of coercions allowed in CAO, we have that $\tau_1 \uparrow \tau_2$ is always uniquely defined. In typing rules, we therefore abbreviate the following pattern

$$\frac{\ldots \quad \Gamma, \Delta \vdash e :: \tau_1 \quad \vdash_{\leq} \tau_1 \leq \tau_2 \quad \ldots}{\Gamma, \Delta \vdash \ldots} \quad \text{by} \quad \frac{\ldots \quad \Gamma, \Delta \vdash e \leq \tau_2 \quad \ldots}{\Gamma, \Delta \vdash \ldots}.$$

Casts. The CAO language includes a cast mechanism that allows for explicitly converting values from one type to another. However, not all casts are possible:

the set of admissible type cast operations has been carefully designed to account for those conversions that are conceptually meaningful in the mathematical sense and/or are important for the implementation of cryptographic software in a natural way. We define a type cast relation $\Rightarrow$, which is associated with a new kind of judgment $\vdash_c$. Table 3 shows the part of the definition of the cast relation. Using the cast relation, we only have to provide one typing rule for cast expressions.

$$\frac{\vdash_{\leq} \tau_1 \leq \tau_2}{\vdash_c \tau_1 \Rightarrow \tau_2} \qquad \frac{\Delta \vdash_t t \leadsto \tau \quad \Gamma, \Delta \vdash e :: (\tau', \mathsf{c}) \quad \vdash_c \tau' \Rightarrow \tau}{\Gamma, \Delta \vdash (t)\ e :: (\tau, \mathsf{c})}$$

The additional rule on the left is needed so that coercions can be made explicit, which also implies that a certain type can be cast to itself.

Sizes of bit strings, vectors and matrices. Since type declarations are mandatory and container types have explicit sizes, we can verify if operations deal consistently with these sizes. Furthermore, the type system can feed this information to subsequent components in the CAO tool chain.

For instance, the operation that concatenates two vectors should return a new vector whose size is the sum of the sizes of the individual vectors, and whose type is the least upper bound of the types of the two vectors, with respect to the coercion ordering $\leq$:

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Vector}[i] \ \mathsf{of}\ \tau_1, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\mathsf{Vector}[j] \ \mathsf{of}\ \tau_2, \mathsf{c}_2) \quad \tau_1 \uparrow \tau_2 = \tau}{\Gamma, \Delta \vdash e_1 \mathbin{@} e_2 :: (\mathsf{Vector}[i+j] \ \mathsf{of}\ \tau, \max(c_1, c_2))}$$

The concatenation of bit strings is similar. Moreover, in the case of matrix algebraic operations, e.g. multiplication, the dimension of the matrices can be checked for correctness.

When range selection is used over bit strings, vectors or matrices, we require that the integer expressions must be evaluated at compile-time so that the size of the expression, and therefore its type can be determined. In this case, the limits of the range are compared against the bounds of the associated type. For instance, for a range access to a vector we have:

$$\frac{\Gamma, \Delta \vdash e :: (\mathsf{Vector}[k] \ \mathsf{of}\ \tau, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j}{\Gamma, \Delta \vdash e[e_1..e_2] :: (\mathsf{Vector}[j-i+1] \ \mathsf{of}\ \tau, \mathsf{c})} k > j, j \geq i \geq 0$$

This is also a limited form of dependent typing since the type associated with the expression depends on the expression itself.

Rings, Finite Fields and Extensions. One of the most unusual features of the CAO language is the support for ring and finite field types and their possible extensions. Our type checking rules allow us to ensure that operations over values of these types are well-defined and that values from different (instances of these) types are not being erroneously mixed due to programming errors. For instance, the typing rule for division is:

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash e_1 :: (\mathsf{Mod}\ [m_1], \mathsf{c}_1) \\ \Gamma, \Delta \vdash e_2 :: (\mathsf{Mod}\ [m_2], \mathsf{c}_2) \quad \mathsf{Mod}\ [m_1] \uparrow \mathsf{Mod}\ [m_2] = \mathsf{Mod}\ [m]\end{array}}{\Gamma, \Delta \vdash e_1 \mathbin{/} e_2 :: (\mathsf{Mod}\ [m], \max(\mathsf{c}_1, \mathsf{c}_2))}$$

The use of the least upper bound captures the fact that the types may be equal, or one may be an extension of the other.

Variables and function calls. The classification of expressions depends on the environment accessed when retrieving the value of a variable. If a local variable is accessed, the code is considered pure; if a global variable is read, the code is classified as read-only.

$$\frac{\Gamma_G(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{ReadOnly})}\ x \in dom(\Gamma_G)$$

$$\frac{\Gamma_L(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{Pure})}\ x \in dom(\Gamma_L)$$

Since in expression, we can only use functions that do not cause side-effects, the typing rule for function application has a side condition to ensure that the body of the function is not a procedure (i.e., it does not modify a global variable):

$$\frac{\begin{array}{c}\Gamma_G(f) = ((\tau_1, \ldots, \tau_n) \to \tau, \mathsf{c}) \\ \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c_n})\end{array}}{\Gamma_G, \Gamma_L, \Delta \vdash f(e_1, \ldots, e_n) :: (\tau, \max(\mathsf{c}, \mathsf{c}_1, \ldots, \mathsf{c}_n))}$$

$$\max(\mathsf{c}, \mathsf{c}_1, \ldots, \mathsf{c}_n) < \mathsf{Procedure} \text{ and } f \in dom(\Gamma_G)$$

Functions, procedures and statements. We introduce symbol $\bullet$ as a possible (empty) return type to detect misuses of the return statement. We distinguish the cases when a block has explicitly executed a return statement from the cases where no return statement has been executed. In the former case we take the type of the parameter passed to the return statement or $\bullet$ if no such parameter exists. In the latter case we also use the $\bullet$ symbol. Thus, a return statement is typed with the same type as its argument, which must coincide with the expected return type for the block.

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\tau_1, \mathsf{cc}_1) \quad \ldots \quad \Gamma, \Delta \vdash e_n \leq (\tau_n, \mathsf{cc}_n)}{\Gamma, \Delta \Vdash_{(\tau_1, \ldots, \tau_n)} \mathsf{return}\ e_1, \ldots, e_n :: ((\tau_1, \ldots, \tau_n), \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma)}$$

Since CAO has a call-by-value semantics, returning multiple values is allowed in order to make references or additional structures unnecessary.

The typing rule for a function definition therefore verifies if the type of its body is not $\bullet$ to ensure that a return statement was used to exit the function. Moreover, the returned type has to be equal (or coercible) to the declared type (recall the use of judgement $\Vdash_\tau$).

The seq statement permits iterating over an integer variable varying between two statically determined bounds. The index starts with the value of the lower (resp. upper) bound and at each step is incremented (resp. decremented) by the amount of the step value until it reaches the upper (resp. lower) bound.

The interesting feature of this mechanism is that the iterator is regarded as a constant at each iteration step. In the typing rules, this allows us to add the index and its respective value to the environment Δ at each iteration:

$$\frac{\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \forall_{n\in\{i\ldots j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$

$$\rho \in \{\tau, \bullet\},\ x \notin dom\ \Gamma_L,\ i \leq j$$

Therefore, declarations and access expressions inside the body of the sequence statement may depend on the index but may still be statically typeable. As highlighted in Section 3, the combination of range selection and assignment operators for bit strings, vectors and matrices with this simplified loop construction is a good example of how the CAO language design allowed us to fine tune the type checker to provide extra assistance to the programmer. Note, however, that sequential statements can make the type checking process slow, as sequences must be explicitly unfolded and typed for each possible value of the iterator.

A Detailed Example. We now present a detailed example of the how our type system handles the hsalsa20 fragment introduced in Section 3. The syntactic form of the program is

```
seq i := 0 to 3 {
    x[i+1]  :=  from_littleendian( k[i*4..i*4+3]);
    x[i+6]  :=  from_littleendian(in[i*4..i*4+3]);
    x[i+11] :=  from_littleendian( k[i*4+16..i*4+19]); }
```

where we desire type annotations for each node in the parse tree. The inference process traverses the tree matching rules against syntax. This traversal highlights aspects of the inference at three levels in the tree. Before reaching this fragment the declarations have already been produced and thus the initial environment is

$$\begin{aligned}
\Gamma_L &= \{\mathsf{k} :: \mathsf{Vec}[32]\ \mathsf{of}\ \mathsf{UBits}[8],\ \mathsf{in} :: \mathsf{Vec}[16]\ \mathsf{of}\ \mathsf{UBits}[8],\ \mathsf{x} :: \mathsf{Vec}[8]\ \mathsf{of}\ \mathsf{UBits}[32]\} \\
\Gamma_G &= \{\mathsf{to\_littleendian} :: \mathsf{UBits}[32] \to \mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8], \\
&\qquad \mathsf{from\_littleendian} :: \mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8] \to \mathsf{UBits}[32]\} \\
\Delta &= \{\}
\end{aligned}$$

The first step matches the entire fragment against seq i := 0 to 3 $\{s_1; s_2; s_3\}$

$$\frac{\forall_{n\in\{0\ldots3\}} \Gamma_G, \Gamma_L[\mathsf{i} :: \mathsf{Int}], \Delta[\mathsf{i} := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ \mathsf{i} := 0\ \mathsf{to}\ 3\ \{s_1; s_2; s_3\} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$

This entails, for each of the $n \in \{0,1,2,3\}$ cases, that for assignments $(l_i{:=}r_i) = s_i$ in each of the s_1, s_2, s_3 preconditions, each statement is matched by

$$\frac{\Gamma_n, \Delta_n \vdash l_i :: (\tau, \mathsf{cl}) \quad \Gamma_n, \Delta_n \vdash r_i \leq (\tau, \mathsf{c})}{\Gamma_n, \Delta_n \Vdash_\tau l_i := r_i :: (\bullet, \max(\mathsf{cl}, \mathsf{c}), \Gamma)}$$

Here $\Gamma_n = \Gamma_G, \Gamma_L[\mathsf{i} :: \mathsf{Int}]$ and $\Delta_n = \Delta[\mathsf{i} := n]$. Now, for each of the l_i we obtain something of the form $\mathsf{x}[\mathsf{i}+1]$ where $\Gamma_L(\mathsf{x}) = \mathsf{Vec}[8]\ \mathsf{of}\ \mathsf{UBits}[32]$ and an index expression $\mathsf{i}+1 :: (\mathsf{Int}, \mathsf{Pure})$, thus we can match

$$\frac{\Gamma_n, \Delta_n \vdash \mathsf{x} :: (\mathsf{Vec}[8]\ \mathsf{of}\ \mathsf{UBits}[32], \mathsf{Pure}) \quad \Gamma_n, \Delta_n \vdash \mathsf{i}+1 \leq (\mathsf{Int}, \mathsf{Pure})}{\Gamma_n, \Delta_n \vdash \mathsf{x}[\mathsf{i}+1] :: (\mathsf{UBits}[32], \max(\mathsf{Pure}, \mathsf{Pure}))}$$

Finally, for each of the r_i the function parameter e_i is either $\Gamma_G[\mathsf{k}]$ or $\Gamma_G[\mathsf{in}]$ :: $\mathsf{Vec}[16]\ \mathsf{of}\ \mathsf{UBits}[8]$, Furthermore, the index expression is defined only over i, whose value is known, and integer literals. Thus each expression of the form $\mathsf{k}[\mathsf{i}*4..\mathsf{i}*4+3]$ becomes a slice over determined indices after application of ϕ_Δ and $\mathsf{k}[\mathsf{i}*4..\mathsf{i}*4+3] :: (\mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8], \mathsf{Pure})$. Hence

$$\frac{\begin{array}{c}\Gamma_G(\mathsf{from\_littleendian}) = (\mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8] \to \mathsf{UBits}[32], \mathsf{Pure}) \\ \Gamma_G, \Gamma_L, \Delta_1 \vdash \mathsf{k}[\mathsf{i}*4..\mathsf{i}*4+3] \leq (\mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8], \mathsf{Pure})\end{array}}{\Gamma_G, \Gamma_L[\mathsf{i} :: \mathsf{Int}], \Delta_1 \vdash \mathsf{from\_littleendian}(\mathsf{k}[\mathsf{i}*4..\mathsf{i}*4+3]) :: (\mathsf{UBits}[32], \max(\mathsf{Pure}, \mathsf{Pure}))}$$

5 Implementation

The CAO type-checker was fully implemented in the Haskell functional language, which provides a plethora of libraries and built-in language features. Among these, we found some to be particularly useful, such as *classes*, specific syntax for handling monadic data types and the *monad Error* data type. These Haskell assets, not only simplified the implementation process, but also helped improving substantially the readability of the code and its comparison with the formal specification of the type checking rules described in the previous section.

To generally illustrate Haskell's ability to deal with the formal type checking rules that we specified in the previous section, consider the following code snippet, which implements the rule for type checking CAO while statements.

```
tcStatement s@(WhileStatement info cond wstms) h rt =
  do (cond', condt, cb) <- tcExp cond h
   checkMatchType info condt Boolean
   (wstms', wst, cc, h') <- tcStatements wstms h rt
   return (mkWhileStatement (buildTcNodeInfo info Bullet)
                            cond' wstms', Bullet, max cb cc ,h)
```

The interpretation of the above code is quite immediate. Function tcStatement is our formal statement type checking function $\Vdash$, rt represents the expected return type, which in the formal definition subscripts $\Vdash$ and h corresponds to the type environments Γ and Δ. Note that, even though we have made clear the distinction between Γ and Δ in the formal rules, this was mainly justified by presentational reasons. Still on the arguments side, one finds (WhileStatement info cond wstms), trivially matching while b $\{c\}$, except for the info identifier, which is an add-on of the implementation for storing the exact place where the CAO code being analysed appears in the input file.

Regarding the function body, in accordance to the formal rule, which relies on premises referring to $\vdash$ and $\Vdash$, so does the implementation, referring to functions tcExp and tcStatements respectively. Here, however, one resorts to Haskell's monadic operator <- over the monad Error data type. In this way we combine calls to different type checking functions that may return type checking errors, ensuring that if an error occurs in one of the calls, the error is propagated down to the end of the type checker execution, without interfering with any other type checking rule in between.

Function checkMatchType corresponds to our order comparison operator $\leq$ over data types, while Bullet is our functional representation of symbol $\bullet$. Function max ensures that type classifiers, which allow the type system to recognise various types of functions, are properly propagated. Instead of returning the type of the expression being evaluated, the implementation returns the expression received annotated with its type, to be used by subsequent compilation steps. Nevertheless, the above rule implementation illustrates how we have kept the implementation reasonably close to the formal definition, therefore favoring a direct validation by inspection of the implementation.

6 Soundness of the Type System

As usual, the CAO type system aims to ensure that *"well-typed programs do not go wrong"* [7]. This is formalised as a soundness theorem relating static (typing) and dynamic semantics. For the moment, our result only ensures that the evaluation of well-typed program does not fall into a certain class of errors: formally, we are proving a *weak soundness theorem.* Concretely, we have shown that only a well-defined set of run-time errors (trapped errors, denoted by ϵ in the semantic domain $\mathbf{V}$) can occur when evaluating a correctly typed program. These are explicitly captured in the semantics of the language, and they are limited to divisions by zero and out of bounds accesses to containers. In this Section, we first shortly present some aspects of our formalization of the CAO semantics necessary to provide support to the subsequent discussion of our soundness theorem and proof sketch. The complete description of both can be found in [1].

CAO Semantics Evaluation of a CAO program is defined by an *evaluation relation* that relates an initial configuration (a CAO program together with a description of the initial state) with a final configuration (a semantic value and a final state). The domain of *semantic values* is defined as a solution of the domain equation $\mathbf{V} = \mathbb{Z} + \mathbf{V}^\star + \mathcal{E}$, where $\mathbb{Z}$ denotes the domain of integers, $\mathbf{V}^\star$ denotes sequences of values of type $\mathbf{V}$ of the form $[v_0, \ldots, v_{n-1}]$ and $\mathcal{E}$ is the type of the trapped error value ϵ. A *trapped error* is an execution error that results in an immediate fault (run-time error); an *untrapped error* is an execution error that does not immediately result in a fault, corresponding to an unexpected behavior. We denote such an error by $\perp$.

We define three mutually recursive evaluation relations, each of them responsible for characterising the evaluation of different syntactic classes: *expressions*, *statements* and *declarations*:

- $\langle\ e \mid \rho\ \rangle \rightarrow r$ evaluates expression e in state ρ to the value r. Expression evaluation is side-effect free, and hence the state is not changed.

- $\langle\ c \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle$ means that the evaluation of statement c in state ρ transforms the state into ρ', and (possibly) produces result r.
- $\langle\ d \mid \rho\ \rangle \Rrightarrow \langle\ \rho'\ \rangle$ means that the evaluation of declaration d in state ρ transforms the state into ρ'.

CAO has a *call by value* semantics, where there are no references and each variable identifier denotes a value. Assignments mean that old values are replaced by the new ones in the state. Since expressions are effect-free, simultaneous value assignments are possible (however, here we will stick to the simpler single-assignment version of the evaluation rule). In CAO, a run-time trapped error can occur only in three cases: 1) accessing a vector, matrix or bit string out of the bounds; 2) division (or remainder of division) by zero; and 3) assigning a value to a vector, matrix or bit string out of bounds. We present example rules for the latter two cases below, noting that the frame update operator is defined to return ϵ when l identifies an update to an invalid index in a container representation.

$$\textsc{Assign-Err}\ \frac{\langle\ e \mid \rho\ \rangle \rightarrow v}{\langle\ l\ \texttt{:=}\ e \mid \rho\ \rangle \Rightarrow \langle\ \epsilon\ ,\ \_\ \rangle}\ \rho[v/l] = \epsilon$$

$$\textsc{Assign}\ \frac{\langle\ e \mid \rho\ \rangle \rightarrow v}{\langle\ l\ \texttt{:=}\ e \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho[r/l]\ \rangle}\ \rho[v/l] \neq \epsilon$$

$$\textsc{Div}\ \frac{\langle\ e_1 \mid \rho\ \rangle \rightarrow v_1 \quad \langle\ e_2 \mid \rho\ \rangle \rightarrow v_2}{\langle\ e_1\ /\ e_2 \mid \rho\ \rangle \rightarrow [\![/]\!][v_1, v_2]}$$

$$\textsc{Div-Zero}\ \frac{\langle\ e_1 \mid \rho\ \rangle \rightarrow v_1 \quad \langle\ e_2 \mid \rho\ \rangle \rightarrow 0}{\langle\ e_1\ /\ e_2 \mid \rho\ \rangle \rightarrow \epsilon}$$

where function at returns the n-th element of a sequence. Range accesses actually cannot cause trapped errors, as the type system enforces that the limits must be statically defined in order to determine the size of the result, which means that such errors can be detected. Trapped errors are propagated throughout evaluation rules, i.e., whenever a premiss evaluates to ϵ the overall rule also evaluates to ϵ. All cases that fall outside of our semantic rules are implicitly evaluated to untrapped errors ($\perp$ value).

Soudness theorem and proof sketch Our result is stated in the following theorem, where $\vdash \rho :: \Gamma_G$ denotes consistency and $\circ$ denotes empty store/state.

Theorem 1. *Given a program p if $\circ, \circ, \circ \vdash p :: (\bullet, \Gamma_G)$ and $\langle\ p \mid \circ\ \rangle \Rrightarrow \langle\ \rho\ \rangle$ terminates, then $\vdash \rho :: \Gamma_G$ or ρ is an error state.*

Proof (Sketch). The full proof is presented in [1]. The proof is by induction on typing derivations. The base case for induction is that prior to execution, every type-checked program has an initial evaluation environment that is (trivially) consistent with the typing environment. Here, consistency means that all variables in the evaluation environment have associated values compatible with their corresponding type in the typing environment. The inductive cases are considered for each transition defined in the semantics of the language. In each case we show that one of two cases can occur: 1) either a consistent environment is produced at the end of each transition; or 2) a trapped error has been generated and is returned by the program. We present two cases, illustrating how the proof proceeds for division expressions and assignment statements that may raise trapped errors.

Division Expressions. We have to prove that if $\langle\ e_1\ /\ e_2 \mid \rho\ \rangle \rightarrow v$ terminates then $v \in \mathbf{V}$. Two semantic rules can be applied for each operator, one in the case of division by 0; the other in the general case:

– If $\langle\ e_1 \mid \rho\ \rangle \rightarrow v_1$ and $\langle\ e_2 \mid \rho\ \rangle \rightarrow 0$ terminate, then $\langle\ e_1/e_2 \mid \rho\ \rangle$ evaluates to $\epsilon \in \mathbf{V}$ by semantic Div-Zero.
– If $\langle\ e_1 \mid \rho\ \rangle \rightarrow v_1$ and $\langle\ e_2 \mid \rho\ \rangle \rightarrow v_2$ terminate, with $v_2 \neq 0$, then $\langle\ e_1/e_2 \mid \rho\ \rangle$ evaluates to $[\![/]\!][v_1, v_2]$ by semantic rule Div. Here $[\![/]\!]$ gives the interpretation of the / operator with respect to the values v_1 and v_2. By induction hypothesis, v_1 and v_2 are in the semantic domain $\mathbf{V}$, corresponding to representations of integer values. Since division is well-defined for integer representations, then $[\![/]\!][v_1, v_2]$ evaluates to another value v which is again a representation of an integer and $v \in \mathbf{V} \backslash \mathcal{E}$.

Assignment Statements. We have to prove that if $\langle\ l := e \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then, either the statement raises a trapped error due to an invalid access on the left value, or the returned environment ρ' is consistent with the typing environment. Two semantic rules are applicable, Assign and Assign-Err, the latter only when the target is an invalid position in a container. If $\langle\ e \mid \rho\ \rangle \rightarrow v$ terminates, then $v \in \mathbf{V} \backslash \mathcal{E}$ and v represents a value of type τ. The semantic rule to apply depends on the result of the frame update operation $\rho[v/l]$. If this returns ϵ, then semantic rule Assign-Err is applied, and the statement evaluates to $\langle\ \epsilon\ ,\ \_\ \rangle$. Otherwise it will return an updated state ρ', in which case semantic rule Assign is applied, and the statement evaluates to $\langle\ \bullet\ ,\ \rho[v/l]\ \rangle$. It remains to prove that this resulting evaluation environment is consistent with the typing environment. Here we resort to the induction hypothesis $\vdash \rho :: \Gamma$, which guarantees the value currently stored for l represents a value of type τ. Since v also represents a value of type τ, the update of left value l for value v preserves consistency.

7 Related Work

Cryptol [5] is a domain-specific language and tool suite 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. Furthermore, CAO tools are released under an open-source policy.

Dependent types offer a powerful approach to ensure program properties. However, this power in not incorporated in any of the mainstream languages, while the prototypical languages that do it are mostly functional. The first prototype of an imperative language to use dependent types was Xanadu [9], allowing, e.g., to statically verify that accesses to arrays are within bounds. So far, CAO offers a modest form of dependent types, where all type parameters values must be statically known. Ongoing work aims extend CAO with a more powerful approach to dependent types inspired by [9]. This new version of the type system allows for symbolic parametrisation, dropping the requirement that all sizes are known at compilation, using an SMT solver to handle associated constraints.

The use of Generalized Algebraic Data Types (GADTs) in Haskell, together with type families and existential types, allows the implementation of embedded DSL's with some dependent typing features. Moreover, since this approach relies on Haskell's type system, this permits avoiding the full implementation of a type checker. CAO does not follow this embedded approach because it would make it harder to preserve characteristics of the language that pre-dated formal work on the type system. For example, the CAO syntax tries to follow the cryptographic specification standards, and GADTs would impose their own syntax, which is more suitable for building combinator systems. One could of course try to use a GADT-based intermediate representation, but it is not clear that this would pay out in terms of the global implementation effort. In particular, we anticipate that dealing with coercions and casts would complicate the type checking apparatus [8]. Moreover, it would probably be difficult using an embedded approach to keep the implementation structure close to the formal specification.

The use of an embedded implementation in a dependently typed language, e.g. Coq or Agda, could also be an option for the implementation of our type system. However, this would suffer from the same drawbacks previously presented for GADTs, and would also require specific expertise that are not realistic to assume in the target audience for CAO. The need to reason about the correctness and termination of CAO programs at this level would also be an overkill for most applications. In the CAO tool-chain, this sort of analysis is enabled by an independent deductive formal verification tool called CAOVerif.

8 Conclusion

CAO is a language aimed at closing the gap between the usual way of specifying cryptographic algorithms and their actual implementation, reducing the possibility of errors and increasing the understanding of the source code. This language offers high-level features and a type system tailored to the implementation of cryptographic concepts, statically ruling out some important classes of errors. In this paper, we have presented a short overview of CAO and the specification, validation and implementation of a type-system designed to support the implementation of front-ends for CAO compilation and formal verification tools.

References

1. M. Barbosa, A. Moss, D. Page, N. F. Rodrigues, and P. F. Silva. Type checking cryptography implementations. Technical Report DI-CCTC-11-01, CCTC, Univ. Minho, 2011.
2. D. J. Bernstein. The Poly1305-AES message-authentication code. In H. Gilbert and H. Handschuh, editors, *FSE*, volume 3557 of *LNCS*. Springer, 2005.
3. D. J. Bernstein. Cryptography in NaCl, 2009. `http://nacl.cr.yp.to`.
4. J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specification Version 2.1, 2003.
5. J. Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *FMSE '07*, page 41. ACM, 2007.
6. A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1996.
7. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Aug. 1978.
8. P. F. Silva and J. N. Oliveira. 'Galculator': functional prototype of a Galois-connection based proof assistant. In *PPDP '08*, pages 44–55. ACM, 2008.
9. H. Xi. Imperative programming with dependent types. In *LICS*, pages 375–387, 2000.