



# Compiling CAO: from Cryptographic Specifications to C Implementations

Manuel Barbosa    David Castro

HASLab/INESC TEC  
Universidade do Minho

Paulo Silva

Braga, Portugal

April 8, 2014

Grenoble

# Motivation

- › Developing cryptographic software is challenging
- › Performance is usually critical
  - › Many implementations are done directly in assembly
- › Aggressive optimizations must not change the semantics
- › Error prone and time consuming

# CAO Language

- › Started in the CACE project (FP7) in collaboration with Univ. Bristol
- › Domain specific language for core cryptographic components
  - › Hash functions, authentication algorithms, signatures, ...
- › High level features closer to standards
- › Supported by a tool chain to assist development

# CAO Language

- › Main design goals:
  - › Flexible and configurable for a wide range of platforms (machine architecture + operating system + compiler + extra libraries)
  - › Incorporate domain specific optimizations early in the compilation process
  - › Oriented to the implementation of cryptographic APIs

# CAO Features

- › Call by value semantics
- › No input/output support
- › No language construct to dynamically allocate memory
- › Highly expressive native types and operators

# CAO Types

## › Booleans

```
def b1 : bool;  
def b2 : bool := true;
```

# CAO Types

## › Booleans

```
def b1 : bool;  
def b2 : bool := true;
```

## › Integers (arbitrary precision)

```
def i1 : int;  
def i2 : int := 10;
```

# CAO Types

## › Booleans

```
def b1 : bool;  
def b2 : bool := true;
```

## › Integers (arbitrary precision)

```
def i1 : int;  
def i2 : int := 10;
```

## › Machine integers

```
def ri1 : register int;  
def ri2 : register int := 1;
```

# CAO Types

## › Booleans

```
def b1 : bool;  
def b2 : bool := true;
```

## › Integers (arbitrary precision)

```
def i1 : int;  
def i2 : int := 10;
```

## › Machine integers

```
def ri1 : register int;  
def ri2 : register int := 1;
```

## › Bit strings

```
def ubs1 : unsigned bits[32];  
def ubs2 : unsigned bits[4] := 0b0101;  
def sbs1 : signed bits[16];  
def sbs2 : signed bits[8] := 1b01010010;
```

## CAO Types (cont.)

- › Rings or fields defined by an integer

```
def mo1 : mod[5];  
def mo2 : mod[2] := [1];
```

## CAO Types (cont.)

- › Rings or fields defined by an integer

```
def mo1 : mod[5];  
def mo2 : mod[2] := [1];
```

- › Extension fields defined by a type and a polynomial

```
def mp1 : mod[ mod[2] <X> / X**7 + X**3 + 1 ];  
def mp2 : mod[ mod[11] <Y> / Y**2 + 1 ] := [5*Y + 2] * [7*Y+1];
```

## CAO Types (cont.)

- › Rings or fields defined by an integer

```
def mo1 : mod[5];  
def mo2 : mod[2] := [1];
```

- › Extension fields defined by a type and a polynomial

```
def mp1 : mod[ mod[2] <X> / X**7 + X**3 + 1 ];  
def mp2 : mod[ mod[11] <Y> / Y**2 + 1 ] := [5*Y + 2] * [7*Y+1];
```

- › Vectors

```
def v1 : vector[10] of register int;  
def v2 : vector[4] of unsigned bits[2] := {  
    0b00, 0b01, 0b10, 0b11 };
```

# CAO Types (cont.)

## › Rings or fields defined by an integer

```
def mo1 : mod[5];  
def mo2 : mod[2] := [1];
```

## › Extension fields defined by a type and a polynomial

```
def mp1 : mod[ mod[2] <X> / X**7 + X**3 + 1 ];  
def mp2 : mod[ mod[11] <Y> / Y**2 + 1 ] := [5*Y + 2] * [7*Y+1];
```

## › Vectors

```
def v1 : vector[10] of register int;  
def v2 : vector[4] of unsigned bits[2] := {  
    0b00, 0b01, 0b10, 0b11};
```

## › Matrices

```
def m1 : matrix[2, 3] of int;  
def m2 : matrix[2, 2] of mod[2] := {  
    [1], [0], [0], [1]};
```

# Simple Example: Bubble Sort

```
typedef int_vector := vector[10] of int;

def bubble_sort(v : int_vector) : int_vector {

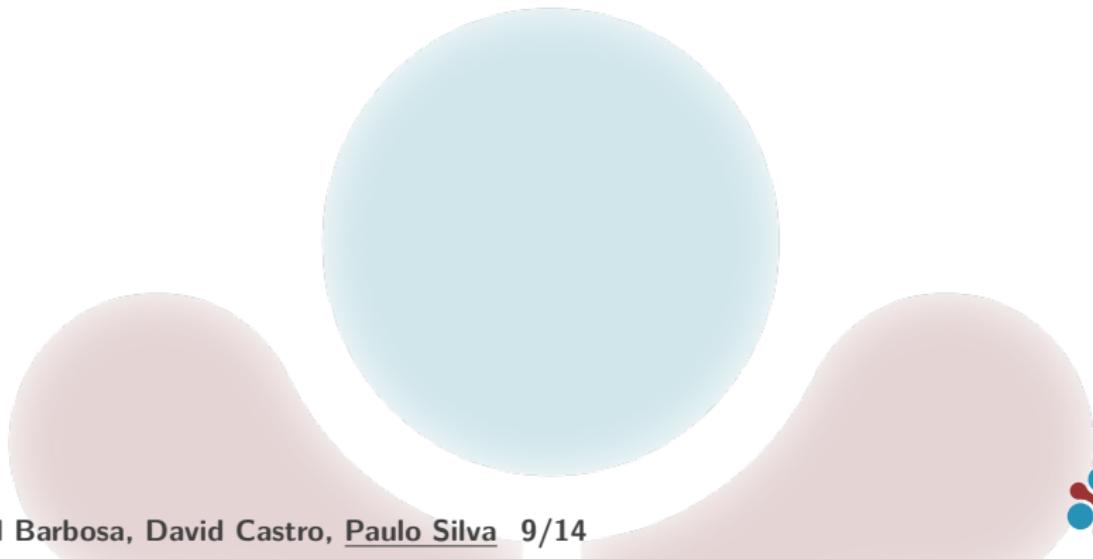
    def temp : int;
    seq i := 8 to 0 by -1 {
        seq j := 0 to i {
            if (v[j] > v[j+1]) {
                temp := v[j];
                v[j] := v[j+1];
                v[j+1] := temp;
            }
        }
    }
    return v;
}
```

# Simple Example: Bubble Sort

```
def bubble_sort(const n : register int {1 < n}, v : vector[n] of int)
: vector[n] of int {
  def temp : int;
  seq i := n - 2 to 0 by -1 {
    seq j := 0 to i {
      if (v[j] > v[j+1]) {
        temp := v[j];
        v[j] := v[j+1];
        v[j+1] := temp;
      }
    }
  }
  return v;
}
```

# Complete Algorithm: SHA1

› (example sha1.cao)

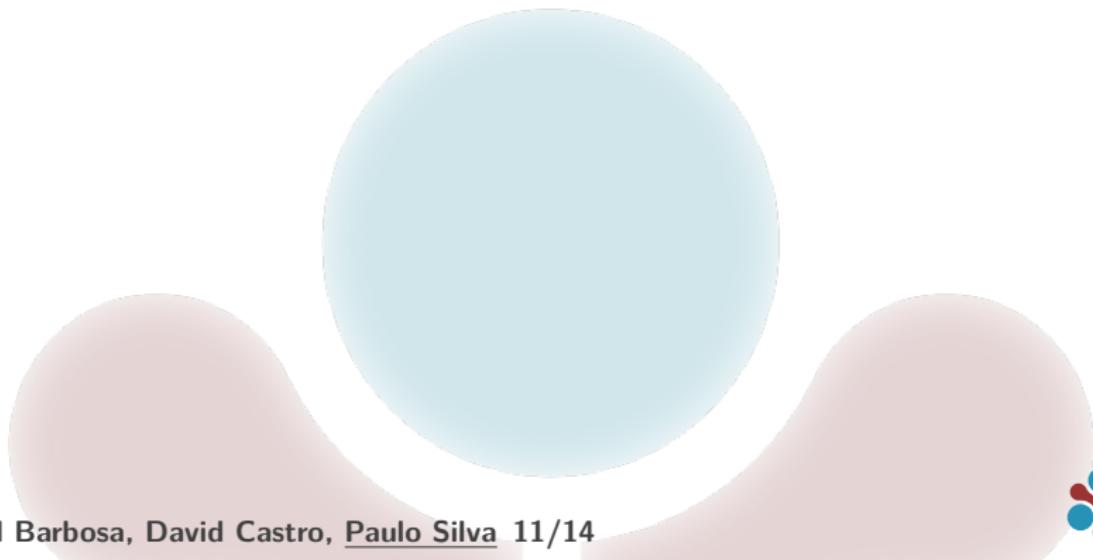


# Exploring Intermediate CAO Code

- › Source to source transformations
- › (demo)

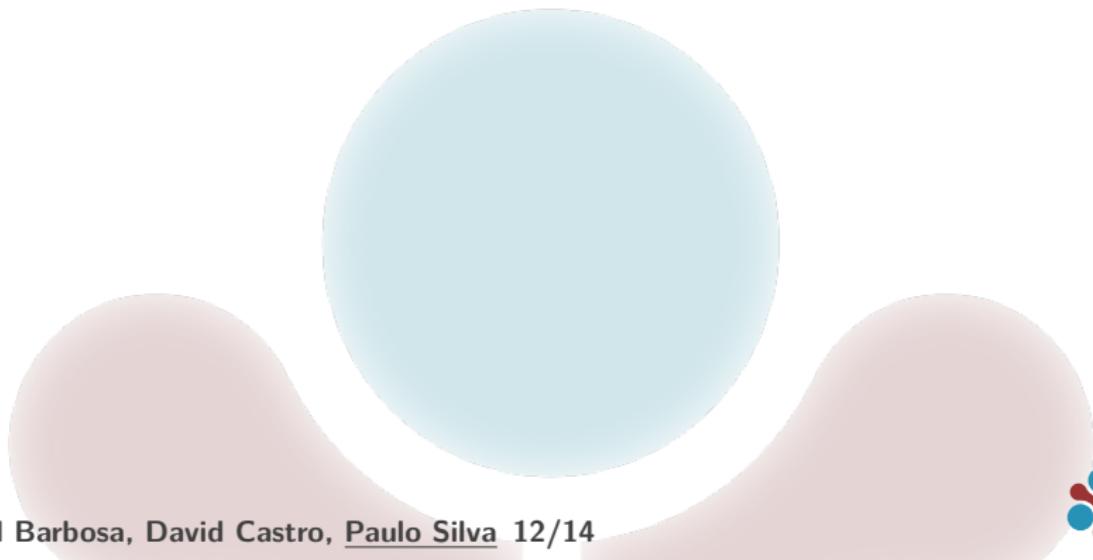
# Platform Specification

› (demo)



# Using the Generated Code

} (demo)



# Protection Against Side-channel Attacks

- › Popular countermeasure against side-channel attacks
- › Indistinguishable functions:
  - › Vulnerable functions execute the same sequence of native CAO operations
- › (demo)

# Conclusions

- › The code of the compiler is reasonably stable
- › The source code is available from the Hackage repository:  
<http://hackage.haskell.org/package/cao>
- › Future work:
  - › Improve efficiency of the generated code (more aggressive optimizations are possible)
  - › Additional protection countermeasures against side-channel attacks
  - › Provide support for other platforms (ongoing work for ARM architecture)
  - › Provide additional guarantees when compiling C using CompCert (ongoing work)