# A Domain Specific Language for Cryptography[*]

Giovanni Agosta
Politecnico di Milano
agosta@elet.polimi.it

Gerardo Pelosi
Università degli Studi di Bergamo
gerardo.pelosi@unibg.it

## Abstract

In this paper, we propose a domain specific language for the development of hardware/software cryptographic systems based on the well known Python programming language. It is designed to allow a wide range of different abstraction levels, and to support native constructs and data types of the cryptography domain, thereby enabling a smoother transition between the cryptographic algorithm design and the target platform implementation phases, as well as an easier automation of the latter phase. The ease of embedding/extending Python in C, C++, and even SystemC makes it a good candidate for the uppermost levels of abstraction in the design flow for this application domain.

## 1 Introduction

Cryptographic subsystems are becoming more and more common in both software and hardware systems, due to the increased emphasis on security requirements in communications and data storage. Cryptographic algorithms development and implementation on specific target platforms is a complex task that requires the cooperation between designers with widely different expertise domains – crpytography and hardware/software systems design. Therefore, development using a general purpose language such as C, with the help of adhoc simulation tools, does not allow all the knowledge available to the cryptographer to be passed to the hardware/software designer. Moreover, language restrictions (e.g., aliasing) may force the hardware/software designer to make conservative choices that could have been avoided with a deeper understanding of the algorithm.

[*]This work was carried out under partial financial support of the Italian MiUR (Project PRIN 2006 ID-2006099978) and in part by project FSE ID-413174.

The above gap can be bridged with the help of an appropriate *Domain Specific Language* (DSL). According to [6], a DSL should offer the following benefits over general purpose languages:

- Domain-specific notations based on established syntax, e.g. from the mathematical representation of algorithms.
- Direct incorporation of abstractions native of the target domain (in our case, e.g., finite fields elements).
- Improved options for automatic transformation (including synthesis, verification, and parallelization).

The major decision patterns [6] that lead to the development of a DSL in the case of cryptographic applications include:

**Notation** Cryptographic algorithms are strongly rooted in Number Theory and Algebra, to guarantee their security. They are therefore more concisely and formally described if the paraphernalia of those mathematical theories are available. On the other hand, applied cryptography requires that such descriptions be still easy to understand for software engineers and architectural designers who may not have advanced expertise in Algebra.

**AVOPT** Analysis, verification, optimization, parallelization and transformation are all paramount to the effectiveness of the porting process of cryptographic system to specific hardware/software platforms; a DSL can help in these tasks by embedding explicit semantic information into the language syntax (e.g., by providing specialized operator for optimizable operations, which would not be easy to automatically detect if they were described as generic functions).

**Data structure representation and traversal**
Cryptographic applications make a heavy use of data types not commonly found in general purpose programs and systems, such as multi-precision numbers, univariate polynomials, and compositions of polynomials, each managed in the arithmetic native to the finite field selected. A DSL language can provide syntactic means to describe and traverse these data in a natural way, while at the same time supporting AVOPT processes.

The above considerations provide a motivation for our proposal of a domain specific language based on the well known Python programming language. The proposed DSL is designed to allow a wide range of different abstraction levels, thereby enabling a smoother transition between the cryptographic algorithm design and the target platform implementation phases, as well as an easier automation of the latter phase. The availability of bindings between Python and C, C++, as well as SystemC [11] allows a natural transition to the lower phases of design refinement and synthesis.

The rest of this paper is organized as follows: Section 2 describes the proposed DSL for cryptographic systems development, using the TEA [12] cryptographic algorithm as a minimal example, while Section 3 introduces a larger case study, using the Blowfish algorithm [9]. Section 4 compares the proposed language with existing or work-in-progress solutions from the literature. Finally, Section 5 draws the conclusions and outlines the future developments of the work depicted in this paper.

## 2 Language Definition

The language is based on Python [8], a dynamically strongly typed language. Several basic features of the language, including functions and iterative constructs are modeled on the Python equivalents – though our proposed language includes the possibility to statically specify data type constraints for variables and parameters.

The current implementation of the proposed language is a simple translator to Python itself, which has allowed a quick development cycle for the prototype and the case studies exibited in Section 3 and 3.

### 2.1 Data Types

Cryptographic algorithms rely heavily on the intractability of several number theory problems to guarantee security. Therefore all public key algorithms need to represent the elements of algebraic structures such as groups, fields and rings in a compact way.

To support the representation of these elements, our language, rather than adopting a set of specialized types, aims at providing a flexible type system that allow to easily write down high-level specifications for the target algorithms.

To this end, our language implements both unlimited precision and fixed precision data types, by allowing each data type to be specified by a size extension. The basic data type is `int`, a signed integer of unlimited precision. Several specifiers can be added to obtain unsigned integers (`u.int`) or fixed precision types (`int.32`). A shortcut for `int.1` is also provided (`bit`), to ease bit manipulations.

It is possible to build user-defined data types by the traditional array construct: e.g., `u.int.32 [4]` defines an array of four 32-bit unsigned integers. Explicit type casting allows array types to be converted into integers of appropriate precision, e.g., an `u.int.32 [4]` array can be converted to an `u.int.128`, using the big endian convention. The reverse is also possible. In these conversion, the actual values are never changed and the total bit size of the data is also unchanged. Casting with semantic changes (e.g., unsigned to signed), widening or shrinking are also allowed.

To support modular arithmetic, a type specifier `mod` is introduced. If a type $t$ is specified as modular, e.g., `mod 5 u.int.32`, an integer value cast to $t$ is guaranteed to be compatible with the residue of the specified modulus – e.g., `(mod 5 u.int.32) 7` is equal to 2.

For applications in cryptography, as an element of an algebraic structure it is possible to consider a polynomial of arbitrary degree with coefficients defined as modular integers or other polynomials. Therefore, it is necessary to provide means to declare *polynomial data types*. In the proposed language, polynomials are declared as arrays of a modular base or user-defined type, with an added `polynomial` specifier. E.g., a `polynomial mod 5 int [3]` can be used to declare a polynomial variable having values of the type $a_2x^2 + a_1x + a_0$, where $a_i \in \{0, 1, 2, 3, 4\}$. A more significant example is `polynomial mod (polynomial mod 2`

Table 1: Scalar Operations.

| Keyword | Operation |
|---|---|
| `+` | addition |
| `-` | subtraction |
| `*` | multiplication |
| `%` | remainder |
| `/` | quotient |
| `**` | exponentiation |
| `!` | negation |
| `&` | logic product |
| `\|` | logic sum |
| `^` | exclusive or |
| `$` | S-box |

Table 2: Vector Operations.

| Keyword | Operation |
|---|---|
| `>>` | shift right |
| `<<` | shift left |
| `[ ; ]` | concatenation |
| `[ , ]` | array construction |
| `>>>` | rotate right |
| `<<<` | rotate left |
| `[ ]` | array element access |
| `[ : ]` | subarray selection |
| `\` | array replication |
| `'` | transposition |

`bit [129]) bit [128]` can be used to declare an element type belonging to the finite field $GF(2^{128})$ with a 128-degree irreducible polynomial, which is part of the definition of Advanced Encryption Standard (AES) [2], one of the most commonly employed block cyphers.

## 2.2 Scalar Operators

To guarantee the basic functionality, the language includes the typical set of scalar operator shown in Table 1.

`$` represents a substitution operation (*S-box*) applied as `X$T` where `X` is `u.int.X` and `T` is array of `u.int.Y [2**X]`.

Note that bitwise operations are not needed since explicit cast is used to perform bitwise operation by reading integers as bit arrays.

## 2.3 Vector Operators

Vector operations are needed to work with arrays and polynomials.

The \ operator performs the replication of array elements, e.g. `v\2` gives `[1,2,1,2]` if `v` is `[1,2]` – it is therefore a shortcut for `[v;v]`.

The concatenation operator differs from the array construction operator in that the two operand arrays are concatenated, rather than used as elements of a new array, so, using the same `v` as above, `[v,v]` yields `[[1,2],[1,2]]`, i.e., a $2\times2$ matrix.

Shifts and rotations are element-wise operations – though if the elements are bits, the semantics of traditional shifts and rotations applied to scalar integers are obtained.

Finally, all scalar operators can be applied to vectors, both arrays and polynomials. They are applied in a coefficient-wise fashion, so applying `(u.int.32)((bit [32])a) & (bit [32])b))` gives the bitwise logic product of the two operands (assumed to be 32-bits unsigned integers). When binary scalar operators are applied to polynomial operands of different length, the lowest degree polynomial is expanded to the size of the highest degree by zero-padding on the most significant elements.

## 2.4 Functions

The definition of functions is preserved from the Python language, with the added option of imposing type constraints to parameters. Figure 1 shows an application of this mechanism to TEA [12], a symmetric key cryptographic algorithm specifically designed for low-end embedded implementation. The rationale for such a choice is to make the language better able to cope with lower-level specifications, where the definition of data type sizes can make a difference in implementation choices. For example, in the synthesis of the algorithm to a target platform, data of small size may fit into architectural registers, while larger data may be split over several registers, or special registers may be added in the architecture.

In the current interpreted implementation of the language, type constraints of parameters are imposed by means of Python decorators (`@accepts` or `@returns`), to which the type constructs are translated before execution.

In addition to standard functions, the proposed language provides a syntactic shortcut to define permuta-

Figure 1: The TEA algorithm implemented in the proposed language

```
def F( u.int.32 v, u.int.32 [2] k, u.int.32 delta) -> u.int.32 :
  return v<<4 + k[0] ^ v + delta ^ v>>5 + k[1]

def code(u.int.32 [2] v, u.int.32 [4] k) -> u.int.64 :
  u.int.32 tot = 0; u.int.32 delta = 0x9e3779b9
  for i in range(64) :
    tot=tot+delta
    v = v[1] , v[0] + F(v[1], k[u.int.2(i)<<1:u.int.2(i)<<1+2], tot)
  return v
```

tions. Permutations are frequently employed as basic blocks of any symmetric cryptosystem. They are generally lengthy to encode in general purpose languages. The proposed syntax makes it more immediate, by allowing the permutation to be defined in terms of its cycles: `perm p :  [3,2,1]`; applied to an array of three elements `a = [x,y,z]`, it has the effect to reorder its elements: `p(a)` $\rightarrow$ `[z,y,x]`.

Some built-in functions are especially useful in cryptographic applications. In addition to the Python-derived built-in functions (`filter` is especially useful in this context), our proposed language provides a pair of functions for random number generation. While the seed generation function has no special features, the `rand` function takes a type definition as parameter, and returns a random element of that type. A shortcut for a random bit is also provided with the keyword `?`.

## 3 Case Study: Blowfish

Blowfish [9] is one of the fastest block ciphers available with no known practical cryptanalysis attacks. It is also in the public domain, which allows free implementations, such as in GnuPG. Blowfish also has several hardware implementations [5, 3], making it a suitable test case for the proposed DSL.

To prove the flexibility of the proposed DSL, Figure 2 shows an implementation of the Blowfish kernel. The encipher function takes as input the plaintext `X` and a subkey array `P` as well as a Feistel function `F`. `F` is generated by the higher-order function `gen_F`, as it is parameterized with respect to a second subkey array, in the form of S-boxes (non-linear functions expressed as lookup tables). DSL features are used to express both the precision of parameters and the type of access to data (by casting 64-bit integers to arrays of 32-bit integers, e.g.) as well as the specificity of S-boxes (via the operator `$`), while high level language features are used to make the implementation readable and concise (a C implementation of the same code is about twice as long).

## 4 Related Works

There are very few proposals in the field of domain specific languages for cryptographic applications.

Cryptol and $\mu$Cryptol [4, 7] are the first commercial attempt at producing languages to speed up the development, optimization, and securing of cryptographic algorithms. They focus on retargetability and validation, but misses the goal of providing the means to describe in a concise and easily comprehensible way the basic operations at the heart of a cryptographic algorithm. As an example, Figure 4 reports the $\mu$Cryptol implementation [10] of the TEA algorithm. Compared to both the C implementation shown in Figure 3 and the implementation in our proposed language described in Section 2, it is not only lengthier, but also much harder to understand for a non-specialized user. Some syntactic elements, such as function definitions, are hard to spot even for the expert user, since they differ widely from the typical representation of such elements in common programming languages, thus making the learning curve steeper.

CAO [1] is a work in progress, aiming at the development of a language and related compilation tools for asymmetric cryptographic protocols and algorithms. CAO is based on C, with some elements from Occam (for management of explicit parallelism) and hardware description languages. With respect to this approach, a Python-based language can capitalize the advantages of its parent language in terms of list syntax expressivity, as well as high-level language features such as higher-order functions (used for example in the Blowfish case study) or metaclasses (used in the current implementation to provide the fixed precision type system as a replacement of the standard Python types).

Figure 2: The Blowfish algorithm implemented in the proposed domain specific language

```
def gen_F(u.int.32 [256] [4] S):
def F(u.int.32 x) -> u.int.32:
    u.int.8 [4] bs = u.int.8 [4](x)
    u.int.32 y = (bs[0] $ S[0]) + (bs[1] $ S[1])
    y = y ˆ (bs[2] $ S[2])
    return y + (bs[3] $ S[3])
return F

def Blowfish_encipher(u.int.64 X, u.int.32 [N + 2] P, F) -> u.int.64 :
   Xl, Xr = u.int.32 [2](X)
   for i in range(N+1) :
      Xl = Xl ˆ P[i]
      Xl, Xr = F(Xl) ˆ Xr, Xl
   return Xr ˆ P[N + 1], Xl ˆ P[N]
```

Figure 3: The TEA algorithm implemented in C language

```
typedef unsigned long word;
void code(word* v, word* k) {
  word y=v[0], z=v[1];
  word sum=0, delta=0x9e3779b9;
  int n=32;
  while (n-- > 0) {
    sum += delta;
    y += (z<<4)+k[0] ˆ z+sum ˆ (z>>5)+k[1];
    z += (y<<4)+k[2] ˆ y+sum ˆ (y>>5)+k[3];
  }
  v[0]=y; v[1]=z;
}
```

Figure 4: The TEA algorithm implemented in the $\mu$Cryptol language

```
exports code;

N = 32;
W = 32;
Word = BˆW;
Block = Wordˆ2;
Key = Wordˆ4;
Index = BˆW;

delta : Word;
delta = 0x9e3779b9;

  code : (Block, Key) -> Block;
  code ([v0, v1], k) = [ys@@N, zs@@N] where {
    rec sums : Wordˆinf;
    sums = [0] ##
           [ sum + delta | sum <- sums ];
    and ys : Wordˆinf;
    ys = [v0] ##
         [ y+((z<<4)+k@0 ˆˆ z+sum ˆˆ (z>>5)+k@1)
         | sum <- drops{1} sums
         | y <- ys
         | z <- zs ];
    and zs : Wordˆinf;
    zs = [v1] ##
         [ z+((y<<4)+k@2 ˆˆ y+sum ˆˆ (y>>5)+k@3)
         | sum <- drops{1} sums
         | y <- drops{1} ys
         | z <- zs ];
 };
```

## 5 Conclusion

In this paper, we have discussed a domain specific programming language for the development of hardwar/software cryptographic applications. The proposed language is based on the well known Python programming language, allowing a fast language development and prototyping cycle, as well as ensuring the grounds for interoperability with C/C++-based languages, thanks to the embedding and extension features of Python.

Future developments will go towards a complete implementation of the proposed language using static compilation techniques, targeting the generation of both C language software implementations and SystemC hardware/software models from the high-level algorithm specification.

## References

[1] M. Barbosa, R. Noad, D. Page, and N. Smart. First steps toward a cryptography-aware language and compiler. Technical Report 2005/160, Cryptology ePrint Archive, 2005.

[2] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[3] I. Gonzalez and F.J. Gomez-Arribas. Ciphering algorithms in MicroBlaze-based embedded systems. *IEE Proc. Computers & Digital Techniques*, 153(2), 2006.

[4] Jeffrey R. Lewis and Brad Martin. Cryptol: high assurance, retargetable crypto development and validation. In *2003 Military Communications Conference (MILCOM 2003)*, volume 2, pages 820–825. IEEE, Oct 2003.

[5] Michael C.-J. Lin and Youn-L. Lin. A vlsi implementation of the blowfish encryption/decryption algorithm. In *ASP-DAC*, pages 1–2. ACM, 2000.

[6] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[7] Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 1–10, New York, NY, USA, 2006. ACM Press.

[8] The Python Programming Language. http://www.python.org.

[9] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In Ross J. Anderson, editor, *Fast Software Encryption*, volume 809 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 1993.

[10] Mark Shields. A language for symmetric-key cryptographic algorithms and its implementation. http://www.cartesianclosed.com/pub/mcryptol/, Jan 2006.

[11] Jol Vennin, Stphane Penain, Luc Charest, Samy Meftali, and Jean-Luc Dekeyser. Embedded scripting inside SystemC. In *Forum on Specification and Design Languages, FDL'05*, Lausanne, Switzerland, September 2005.

[12] David J. Wheeler and Roger M. Needham. Tea, a tiny encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 363–366. Springer, 1994.