Non-Reversible VHDL Source-Source Encryption

Kevin O'Brien, Serge Maginot
LEDA S.A. 35 Avenue du Granier 38240 Meylan France
Tel: (+33) 76 41 92 43     Fax: (+33) 76 41 92 44

Abstract

In this paper, we present KRYPTON, a non-reversible, VHDL source to source encryption tool. By generating non-reversible encrypted VHDL we do away with the classic problems of encryption tools such as portability and the risk of decryption. KRYPTON has been successfully tested on industrial models containing over 100 VHDL units.

1. Introduction

Due to the ever-increasing cost of IC development, microelectronics companies now find themselves in the unfamiliar position of having to co-operate with one another to reduce both costs and time to market. This implies however, that they will inevitably have to share design-related information, some of which may be confidential. Therefore, a way has to be found whereby this information can be exchanged without revealing its content. Unfortunately, the most suitable means of transferring and sharing data, guaranteed to work on different platforms, is with a standard description format such as VHDL. The problem remains however, of how to ensure the confidentiality of this data as descriptions in VHDL are relatively easy to interpret compared to other, less portable forms of transfer such as binary code.

In this paper, we examine a means of encrypting VHDL code that maintains the portability of VHDL, as the output of the encryption tool is also VHDL, but ensures the confidentiality of this code by generating unintelligible, non-reversible VHDL. By unintelligible we mean not only unreadable to the human eye but also unrecoverable by smart editors or lexical analysers. In addition, the original code can never be regenerated due to the use of non-reversible transformations. This form of encryption is made possible by the use of an internal schema representation of a VHDL model generated by a full-VHDL compiler [4].

Let us look at a typical example where two microelectronics companies have decided to partner on a once-off basis in order to combine their two models on a chip. A typical model consists of a large number of hierarchical entity-architecture pairs that interface with a set of packages. These packages may be used for describing the environment surrounding the model such as a library of units representing the underlying technology or some common functions, as well as standard packages such as std_logic_1164. The partner supplying this model would like the recipient to be able to simulate it but not to be able to use the information supplied elsewhere.

To facilitate the "lending" of such information while at the same time maintaining a necessary amount of secrecy, we have developed an encryption tool, KRYPTON, that takes as input a full VHDL model and generates a functionally equivalent but encrypted model, also in VHDL.

KRYPTON, is a non-reversible VHDL source to source encrypter. By non-reversible we mean that some of the original information (such as identifier names) is irretrievable, even to the partner generating the encrypted source. This is in contrast to other encryption tools that facilitate the release of pre-compiled object files. In addition, KRYPTON generates fully compilable and simulatable VHDL code (either VHDL'87 [1] or VHDL'93 [2]) which implies much greater portability and security than crypters generating binary code. Generating encrypted VHDL is more portable as suppliers of binary code must provide a different binary version for each platform (and each version of each platform) on which the code will be executed. From the clients point of view, having VHDL source code means not having to waste time contacting the supplier for a new version each time there is an in-house upgrade in platform. Encrypted VHDL is more secure as it is possible to reverse analyse the binary code on some platforms.

Obvious transformations include modifying all identifier names thereby rendering them meaningless [3]. It is, of course, necessary to preserve those names used for simulation. With KRYPTON, the user can specify that the top units of the model or even individual identifiers are not to be encrypted. The renaming algorithm also
maximises the overloading of subprogram and parameter names, thus making the encryption truly non-reversible, even by lexical analysers. Another encryption transformation that reduces the reusability of the model is the removal of unused declarations. This removal should be tightly controlled however as it may not always be desirable. Finally, the output of KRYPTON is both compilable and simulatable.

In the rest of this paper, we will detail KRYPTON, its surrounding environment and the main transformations carried out. In the next section, we give a brief description of some of these transformations. In section 3, we show the need and the benefit of using a compiler with an accessible intermediate format as a front end to KRYPTON. Section 4 describes the tool itself and presents a full example. Finally, section 5 concludes the paper and presents some perspectives for the future.

2. Encryption transformations

The main objective of KRYPTON is to take a VHDL model as input and to generate an unintelligible but functionally equivalent VHDL model as output. In order to render the model unintelligible, KRYPTON applies several different transformations to the original source. This is done by accessing an intermediate schema of the model generated by the LEDA VHDL System (LVS) compiler [4]. The benefits of using such a system will be shown in section 3. In this section, we assume that the model has been fully compiled.

To render the model unintelligible many different transformations are applied. For the model to be unreadable by the human eye, all names are scrambled and made to resemble each other, indentation, formatting and comments are removed [3]. To prevent decoding by smart editors and lexical analysers, subprogram names (and parameter names) are overloaded, locally static expressions are replaced by their values and unused objects, types, subprograms, constants and attributes are removed. All of these features are optional, allowing a tight control of the encryption process. For example, if a specific signal is required for simulation, the user can indicate that this signal, and only this signal, is not to be encrypted. Another important fact is that none of the transformations applied by KRYPTON will affect the speed of simulation.

KRYPTON performs the encryption of a model by making several passes over the schema representation and on each pass executing a different set of transformations. The order of execution of these transformations is important as they may be interdependent. For example, it is preferable to rename identifiers after those that are unused have been removed. The order of the main transformations carried out by KRYPTON is as follows:

• Preprocessing
• Locally static expressions are replaced by their value
• Unused constants, subprograms, objects, types and attributes are removed
• Identifiers are renamed and overloaded where possible

2.1 Preprocessing

In a first pass, KRYPTON scans all units performing some minor transformations and collecting information that is necessary for subsequent passes. Two of the transformations applied during this pass are the replacement of all binary and unary operations by equivalent function calls (thereby increasing the possibility of overloading during later passes) and the association of all unbound component instantiations with an entity of the same name, if it exists (thus ensuring that, if the component has a default configuration corresponding to this entity, the encrypted names of both the component and the entity will remain the same). In addition, the last compiled architecture of each entity is noted.

A final task during this pass is to tag all identifiers that are directly or indirectly specified not to be removed or not to be encrypted (see section 4).

2.2 Replacing locally static expressions

The first major transformation to be executed by KRYPTON is to replace all locally static expressions by their values. For example, suppose we have the code:

```vhdl
package P is
  constant MIN : integer := 0;
  constant MAX : integer := 100;
  signal X : integer := MIN + MAX / 2;
  signal Y : integer;
end P;
```

When considering this transformation only, the declaration of signals X and Y becomes:

```vhdl
signal X : integer := 50;
signal Y : integer := integer'left;
```

During compilation, all locally static expressions are calculated. KRYPTON replaces each expression by its calculated value. Some care has to be taken though when initialising integer-type and real-type declarations as not all VHDL compilers agree on the exact ranges of these types (whether they are symmetric about zero or not). To overcome this, KRYPTON uses the `LEFT` attribute as the initial value for these two types.

This type of transformation reduces the possibility of recovering the initial source. It also enables us to remove locally static constant declarations such as MIN and MAX in the next pass.
2.3 Removing unused declarations

The next step is to remove all unused declarations including constant declarations that have been made redundant by the previous transformation. To eliminate all unused declarations, it may be necessary to make several passes over the models, as the next example will show. Suppose we have the following code to encrypt:

```vhd
package P is
    type T is (T1,T2,T3);
    function F(A : integer) return T;
end P;
```

During a first pass, the type declaration \( T \) will not be removed as it is used in subprogram \( F \). However, as \( F \) is never used it can be removed thus paving the way for the removal of \( T \) during a second pass.

The removal of unused attribute specifications also enables models to be rendered unusable by tools requiring synthesis-specific or simulation-specific attributes such as those used by Redwood's VHDL simulator REVEAL [5]:

- `attribute redwood_periodic_clock of...`

   Of course, such transformations are not always desirable and KRYPTON allows the user to tightly control their use.

2.4 Renaming

Those identifiers remaining after the previous transformations are then encrypted and overloaded as much as possible. The encrypted names generated are completely random making it impossible to recover the original source. All encrypted names are made to resemble each other, each using only the characters \( 1,1,1,0 \) and \( 0 \). This can be seen quite clearly in the full example of figure 6.

The encryption process progressively removes the intelligibility and the recoverability of the original VHDL source. The final stage of KRYPTON is to invoke a reverse analyser to generate the encrypted code without any of the original formatting. The result of this can be seen in section 4.

3. Ensuring transformation correctness

Although some of the encryption transformations appear relatively straightforward, to ensure correct encryption it is essential to use an environment that allows all units of a model to be opened simultaneously in an accessible intermediate format.

The LEDA VHDL System (LVS) [4] provides all of these properties. LVS is a compilation environment developed for VHDL-based applications. It compiles VHDL '93 code but also contains a VHDL '87 switch. It allows the analysis of VHDL descriptions and their storing in a binary data format accessed through a procedural interface. The binary format schema (VIF, VHDL Intermediate Format) is fully derived from the IEEE working documents of the VIFASG [6]. Furthermore, the system allows both temporary and permanent user-defined extensions to the schema.

At the conceptual level, the VIF is a tree-like structure whose nodes possess attributes pointing to values of one of three kinds:

- (i) primitive values (integers, enumerated values_)
- (ii) simple links to other nodes
- (iii) lists of links to other nodes

These attributes are accessed and modified through a procedural interface (LPI) that includes an expandable set of functions operating on library units, nodes, attributes and lists as well as allowing the development of error-handling routines and escapes.

To generate functionally equivalent encrypted code, KRYPTON follows the schema from unit to unit to propagate the correct information, including indications that certain units or identifiers have been directly or indirectly earmarked not to be encrypted. Suppose we have the following code as input:

```vhd
package P is
    type XBIT is ('X', '0', '1');
end P;
```

In the input command file, we specify that entity \( E \) is not to be encrypted. This not only means that the identifiers \( E \) and \( Q \) must not be touched, but also the identifiers \( P \) (used in the `use` clause of entity \( E \)) and \( XBIT \) as well as the character literal \( '0' \). In order to pass this information to package \( P \) we use the schema.

The schema representation of entity \( E \) is shown in figure 1. Names in bold represent nodes or lists of nodes, whereas those in italics are attributes of these nodes. From figure 1, we can see that it is possible to follow the appropriate paths in the schema to locate information in other units. For example, to inform the type declaration node for \( XBIT \) that its identifier is not to be encrypted, we must first get the `has_constraint` attribute of the port declaration node and from this identify the subtype. Once we find this, we tag the node corresponding to its identifier's entry in the symbol table.
Once all of this information has been propagated, we can begin encryption. The first major transformation is the replacement of locally static expressions by their value. Again, this is made relatively easy through the use of a schema of the model. Consider the following code:

```
package P is
  constant MIN : integer := 0;
  constant MAX: integer := 100;
  signal S : integer := (MIN + MAX) / 2;
end P;
```

During compilation, LVS evaluates all locally static expressions and stores the result in the schema as shown in figure 2 for the signal declaration.

Note that the first binary operation node also contains the result of the operation in its `has_value` attribute. This information is used to generate a new numeric_value node and this node replaces the entire subtree in the shaded box of figure 2. Let us now consider the case where we want to eliminate unused declarations. KRYPTON must check every unit for references to all declarations and when one is found, it must follow the schema of the model to the node representing the declaration and tag this node indicating that it is not to be removed.

```
signal_decl has_value_exp
```

Figure 2: Replacing locally static expressions

Figure 3 shows some VHDL code used to illustrate how KRYPTON removes unused declarations. The code in figure 3 contains three inter-dependent units. To identify all unused declarations, KRYPTON must first tag those that are used. This is not a trivial task. Take for example the three subprogram declarations in package `P` (lines 2-4). VHDL allows many different ways to call these subprograms. Some of these are illustrated in the rest of figure 4.

1. package `P` is
2.   function RESOLVED(X : bit_vector) return bit;
3.   function TO_REAL(A : integer) return real;
4.   function "+"(A : real, B : integer) return real;
5.   subtype RBIT is RESOLVED bit;
6.   subtype RBIT2 is RBIT;
7.   subtype RBIT3 is RBIT2;
8. end P;
9. use WORK.P.all;
10. entity E is
11.   port(P: in RESOLVED bit; Q: in real);
12. end E;
13. architecture A of E is
14.   signal S1:integer; signal S2:real;
15.   signal S3:RESOLVED bit;
16.   component E port (P: in RESOLVED bit; Q: in real);
17. end component;
18. begin
19.   S2 <= S1 + S2;
20.   L : E port map(P => S3, Q => TO_REAL(S1));
21. end A;
```

**Figure 3: Using subprograms in VHDL**

The first function call appears in line 5 of figure 3. This statement is a resolvable subtype declaration. When KRYPTON identifies this declaration, it follows the path to the declaration as defined by the schema and shown in figure 4(a). In other words, a node corresponding to a resolvable subtype declaration exists in the schema and this node contains an attribute `has_resolve_func` that leads straight to the subprogram declaration. KRYPTON follows this path and tags the declaration of line 2 as used.

Along similar lines, the port declaration node corresponding to `P` has an attribute `has_resolution` that also leads to the declaration node corresponding to line 2. This is also true for signal declaration `S3`. This path is shown in figure 4(b).

Binary and unary operation nodes have an attribute called `has_func_name` that leads to the appropriate function or operator declaration. On line 17 of figure 3, this path leads to the operator declaration of line 4.
KRYPTON must also distinguish implicit operator and function declarations from user-defined ones as the former must not be removed or encrypted.

![Diagram](image)

**Figure 4: Schema generated for different subprogram calls in VHDL**

VHDL also allows function calls to appear as the actual part of an association element. An example of this is shown in the component instantiate statement on line 18 of figure 3. To find and tag the corresponding subprogram declaration, KRYPTON must follow the path shown in figure 4(c).

In the example of figure 3, we note that no function declarations can be removed as they are all used. However, we can remove the subtype declaration RBIT3 (line 7). A subsequent pass will identify the fact that RBIT2 is no longer used and this declaration will be removed. A third pass eliminates the now redundant declaration of RBIT. KRYPTON stops testing for unused declarations once it has made an entire pass without removing a single declaration. Tests with large examples containing over 100 units indicate that five or six passes are usually sufficient.

Once all unused declarations have been removed, KRYPTON renames all remaining identifiers and maximises the overloading of subprogram names and parameters. KRYPTON generates a random but unique name for each identifier remaining in the schema.

One area in which much care must be taken is in the renaming of entity declarations used as default configurations by components. It is imperative that both the entity and the component keep the same encrypted name. This can be done very efficiently through the use of a schema. During an early pass, all unbound component instantiations are noted and a search is carried out for an entity bearing the same name as the corresponding component declaration. If it exists, a pointer to it is stored in the component declaration node. This pointer is destroyed if it transpires that the component is configured elsewhere in the model. During the renaming phase, KRYPTON checks if this pointer exists in each component declaration node. If so, the node is assigned the same name as the entity. If not, a new random name is generated and assigned to the component. The alternative to this is to generate encrypted names based on the original name. This however is non-random by nature and it is therefore possible to regenerate the original name if the encryption algorithm is broken.

### 4. Running KRYPTON

An overview of the environment surrounding KRYPTON is shown in figure 5. The input consists of a set of VHDL files representing the model to be encrypted and a command file that indicates the names of the VHDL files as well as various parameters pertaining to the encryption. The output consists of another set of VHDL files, this time containing the encrypted units, a makefile to compile these units in the correct order and a log file that presents a record of the execution.

The input command file allows the user to fully control the encryption of each unit and permits the user to indicate which declarations not to encrypt, which declarations not to remove and so on.

![Diagram](image)

**Figure 5: The KRYPTON environment**

The output encrypted files are generated with a reverse analyser that transforms the intermediate format into compilable VHDL source code. The reverse analyser can generate VHDL'87 or VHDL'93 code, based on the parameters supplied in the input command file.

An example of an input VHDL file is shown in figure 6. We assume that these units are contained in a file named "example.vhd".

When running KRYPTON, we want to encrypt everything except the declaration of variable \( v \) in process \( p \) (line 27) We also want to see the extent of encryption of package L4_PKG. An outline of the input command file looks like:
We are assuming that all units will be compiled into a library named LIB1. KRYPTON generates three VHDL files corresponding to the three units. These files are shown in figure 7.

Although figure 7 is not very legible, there are some important transformations that should be pointed out. First, the condition of the if statement on line 31 of figure 6 was deemed to be locally static and thus the expression was replaced by its value. This paved the way for the removal the function to_int and the attribute optimized, declared on lines 9 and 10 of figure 6 respectively. Note also that the after clause of line 34 was not evaluated. This is because the units were compiled in VHDL'93 mode and time expressions are not locally static in this mode.

Another point worth noting is that the overloading of subprogram names extends beyond unit boundaries, thus ensuring the maximum of overloading.

KRYPTON has been successfully evaluated with large, real examples including the g721 [7] algorithm specification (containing 26 units) and g722 [7] circuit specification (containing 102 units) supplied by the CNET\textsuperscript{1}. Both encrypted versions give identical simulation results as the original descriptions on several platforms.

5. Conclusions and perspectives

We have presented a VHDL source to source encryption tool, KRYPTON, that enables the encryption of VHDL models so that they remain compilable and simulatable, but their contents are indecipherable. This is achieved by applying non-reversible encryption algorithms to an internal schema representation of the model. This type of encryption is more portable as the output is also VHDL and less prone to attack as some of the transformations are completely irreversible. KRYPTON has been successfully tested on real industrial applications.

We are now looking at ways to add even more encryption, including an option to render the inputs non-synthesizable. Such transformations include the addition of states (i.e. identifying the enumerated type corresponding to the state variable and adding dummy enumerated literals), case alternatives, false branches, wait statements (in false branches) read and write statements and so on.

Acknowledgements

The authors would like to thank Dr. Vincent Olive of the CNET for his contribution to the specification and the realisation of this work.

\textsuperscript{1} Centre National d'Etudes des Télécommunications, 28 Chemin du Vieux Chêne, BP98, 38243 Meylan Cedex, France
package L4_PKG is
  type xbit is ('X','0','1','Z');
  type xbit_vector is array(natural range <>) of xbit;
  function resolved(x : xbit_vector) return xbit;
  function "or"(x,y:xbit) return xbit;
  subtype rbit is resolved xbit;
  type rbit_vector is array(natural range <>) of rbit;
  function "and"(x,y:rbit_vector) return rbit_vector;
  function to_int(x : xbit_vector) return integer;
  attribute optimized : boolean;
  attribute optimized of all : function is true;
end L4_PKG ;
use WORK.L4_PKG.all;
entity L4_ENT is
  port( a,b : in rbit_vector(1 to 10);
       clock: in xbit := '0';
       s : out rbit_vector(1 to 10));
end L4_ENT ;
package O1O01O00I00Ol is
    type I1O00Il1O00Ol1 is (    O0O00Il1Ol0I01I, I0Ol1O01I00Ol1, O1Il1Ol0O01I01, I1Il100000I1     );
type O0Il00000100Ol1 is array  (  NATURAL  range <> ) of I1O00Il1O00Ol1 ;
pure function I0I01Ol0Ol1I  ( constant O1Ol0Il0O00Ol1 : in O0Il00000100Ol1 ) return I1O00Il1O00Ol1I ;
pure function I0I01Ol0Ol1I  ( constant O1Ol0Il0O00Ol1 : in I1O00Il1O00Ol1I ; constant I1Ol1Ol0I00Il : in I1O00Il1O00Ol1I ) return I1O00Il1O00Ol1I ;
type I0O01Ol1Ol0Il1 is array (  NATURAL range 1 to 10 ) of O0Ol1Il1Il01Ol1 ;
pure function I0I01Ol0Ol1I  ( constant O1Ol0Il0O00Ol1 : in I0O01Ol1Ol0Il1 ;constant I1Ol1Ol0I00Il : in I0O01Ol1Ol0Il1 ) return I0O01Ol1Ol0Il1 ;
end package O1O01O00I00Ol ;

use WORK.O1O01O00I00Ol.all;

entity O1I01I00I00I01I  is
    port(signal I1I01I00O00Ol1I:in I0O01Ol1Ol0Il1(NATURAL range 1 to 10) ;signal O0I00I01I00Il1 : in I0O01Ol1Ol0Il1 ( NATURAL range 1 to 10) ; signal  I0Il0O00Il0Ol1 :  in  I1O00Il1O00Ol1I  :=  I0Ol1O01I00Ol1I  ; signal O1O00Ol1Ol0O01I :  out  I0O01Ol1Ol0Il1  ( NATURAL range 1 to 10);end entity O1I01I00I00I01I ;

architecture I1O01Ol1O00Il0I of O1I01I00I00I01I is
begin
    function I0I01Ol0Ol1I return  I1O00Il1O00Ol1I   is   begin
        return  '0' ;
    end function I0I01Ol0Ol1I ;
    function I0I01Ol0Ol1I  ( constant O1Ol0Il0O00Ol1 : in I1O00Il1O00Ol1I ) return  I1O00Il1O00Ol1I   is
        variable I0Ol0O00Il0Ol1I :  I1O00Il1O00Ol1I  :=O0O00Il1Ol0I01I;begin
            return
                WORK.O1O01O00I00Ol.I0I01Ol0Ol1I   (O1Ol0Il0O00Ol1 =>   I0Ol0O00Il0Ol1I   , I1Ol1Ol0I00Il =>  O1Ol0Il0O00Ol1 ) ;
        end function I0I01Ol0Ol1I ;
    begin
        O1Il0O00I00I01I :  process  variable V :I0O01Ol1Ol0Il1  (NATURAL range 1 to 10) := (others => O0O00Il1Ol0I01I    )    ;
        variable I1Il0I00Il1I01 :   I1O00Il1O00Ol1I   :=O0O00Il1Ol0I01I ;
        begin
            wait  on  I0Il0O00Il0Ol1  ;
            if  TRUE  then
                V  := WORK.O1O01O00I00Ol.I0I01Ol0Ol1I
                    (O1Ol0Il0O00Ol1 =>   I1O00Il1O00Ol1I   ,I1Ol1Ol0I00Il =>  O0I00I01I00Il1  ) ;
            end if  ;
            O1O00Ol1Ol0O01I  <=
                inertial V after   2   *   5 NS  ;
                I1Il0I00Il1I01  :=  WORK.O1O01O00I00Ol.I0I01Ol0Ol1I
                        (O1Ol0Il0O00Ol1 =>   I1O00Il1O00Ol1I   '  ( I0I01Ol0Ol0Ol1I  ( O1O01Ol00000Ol1I  ( I1O00Il1O00Ol1I1 )  ) ) , I1Ol1Ol0I00Il =>  I1O00Il1O00Ol1I  ' ( I0I01Ol0Ol0Ol1I  ) );
        end process O1Il0O00I00I01I;end architecture I1O01Ol1O00Il0I ;
end package O1O01O00I00Ol;

architecture L4_ARC of L4_ENT is
begin
    function toto return xbit is
        begin return '0';
    end return;
begin
    function toto(a : xbit) return xbit is
        variable vvv : xbit;
begin
    return vvv or a;
begin
    p: process
        variable v : xbit_vector(1 to 10);
begin
    w := toto(w) or toto;
    end process p;
end
Figure 7: Result of encryption of the three units of figure 6

References