DARXplorer 2.0 – User Documentation

DENNIS HOPPE
Bauhaus-University Weimar

DARXplorer supports the differential cryptanalysis of primitives which use a mixture of addition, rotation and xor. DARXplorer attempts to find good xor differences. The two main use cases for DARXplorer are aiding the designers of new primitives to defend against attacks and analyzing existing primitives to find attacks.

Cryptanalysts are often confronted with new primitives demanding new tools for analyzation. DARXplorer offers a convenient way to describe such a primitive directly in Ada at minimal costs. The comprehensive analyzing framework including extensive logging mechanisms is already at ones feet.

1. INTRODUCTION

DARXplorer was one of the MSL laboratory projects in the summer term 2008. Goal of the project was to create a framework supporting the development of the block cipher *Threefish* used in the hash function *Skein* [Ferguson et al. 2008]. Threefish based on a mixture of addition modulo 2^{64}, rotation and XOR.

Addition, rotation and xor are low-level operations which run very fast on almost all computer platforms. Not surprisingly, many symmetric cryptosystems make intense use of these operations – and some even use these three operations exclusively, e.g., the Helix/Phelix family of authenticating stream ciphers [Whiting et al. 2003; 2005]. As it turned out, the Achilles heel of these primitives was some vulnerability to differential cryptanalysis, mostly based on XOR differences [Paul and Preneel 2005; Wu and Preneel 2006].

When analyzing a given primitive of this type, searching for good xor differences appears to be the way to go. Here, good means that the probability of differential propagation is not too low. On the other hand, when designing a new primitive of this type, one must analyze such attacks. Ideally, the designer should show that the probability of any differential propagation is too low for an attack. In this context, one needs to search for the best xor difference, i.e., for an xor difference with most probable differential propagation. If the probability is just too low, the designer can make a strong point that the primitive is secure against this type of attack. DARXplorer attempts to find good xor differences.

Besides supporting the development of new primitives, DARXplorer's second goal is to find attacks against existing primitives like *SHA-256*. For that reason, DARXplorer was enhanced in a consecutive research project in the summer term 2009 to allow the analyzation of arbitrary primitives so long as they use a mixture of the previous stated mixture of operations.

2. TECHNIQUES TO FIND GOOD DIFFERENCES

DARXplorer implements four different techniques to find "good" characteristics. Usually, the differential behavior of a low Hamming-weight difference is better predictable (i.e., with a higher probability) than the behavior of a "random" difference.

We just start with such a low Hamming-weight difference (1, 2, 3 or 4) and try all possible input differences with the chosen Hamming-weight. The four approaches differ in the way, they pursue differences from round to round.

Lazy Laura: Laura just "guesses" that the the addition behaves like the xor, difference-wise. I.e., there are no carry bits generated. Starting with low Hamming-weight differences, it is very easy for her to compute the output difference (running the primitive in forward direction). The benefit of Laura's approach is that it is very efficient and can be implemented for any number of rounds (for 80 rounds, or for 800 rounds).

Greedy Grete: Laura just assumes no carry bits are generated. This is likely to exclude many "good" characteristics – a carry bit in one round leads to another difference in another round and may ultimately lead to a better (more probable) difference at the end. Grete optimizes locally. Given some round's input differences in some round, she enumerates all the most probable output differences. All of them are considered as the input difference for the next round. If there is more than one optimal difference, Grete's run time will grow exponentially with the number of rounds.

Pedantic Petra: Petra goes a step further than Grete. Given an input difference, she enumerates all output differences with a nonzero probability. Recall that this is feasible as long as the Hamming-weight of the difference is low. As the number of possible differences is growing fast, and the Hamming-weight also grows with the number of rounds, this can be done only for a few rounds. Petra is pedantic, but not stupid. A naive approach would make her deal with a huge number of extremely low-probable characteristics. But Petra sorts her intermediate results (characteristics with less than the desired number of rounds) by their probability. She always proceeds with the most probable intermediate result. Specifically, when she comes to the last round she just uses Grete's approach.

Fuzzy Fiona: Fiona's strategy is a new cryptoanalytical approach. Assuming that the input for a S-Box, i.e. for a non-linear function, is `100101`. The S-Box describes a mapping from 6 bits to 4 bits. However, the output can only be defined exactly for particular bits: `1??0`, whereby question marks represent bits, whose value cannot be clearly fixed. Even so, one can specify the probability, that a bit equals `1`. Now, rewriting the output with probabilities results in $(1, \frac{1}{4}, \frac{2}{3}, 0)$. The third bit takes on value `1` with probability $\frac{2}{3}$, and the last bit is definitly `0` and is never equals `1`. Pursuing such probabilities while computing a hash digest, let one employ predictions about the output, if the probability is distinct from $\frac{1}{2}$.

3. CRYPTOGRAPHIC PRIMITIVES

The `examples` folder contains declarations of cryptographic primitives in Ada, that are based on a mixture of addition, rotation and XOR. It is a good place to get started, if you want to learn, how to write descriptions by yourself. At present, we included the following primitives:

— Blake-32
— Blake-64
— Blue Midnight Wish 256
— Blue Midnight Wish 512
— Edon-R 256
— Edon-R 512
— Salsa-20
— SHA-256
— SHA-512
— Tiny Encryption Algorithm (TEA)
— Threefish-256
— Threefish-512
— Threefish-1024

Both *Blue Midnight Wish* and *Threefish* are also covered with revised versions including changes made by submitting tweaked primitives to the second round of the NIST competition.

In the next chapters you will learn how to write your DARXplorer-adapted version of a cryptographic primitive, verify it and analyze it.

4. A SMALL EXAMPLE

Suppose you want to integrate *TEA* in DARXplorer. Your source file for encryption could look like this:

```
1  with DXPL_Types_32; use DXPL_Types_32;
2
3  procedure TEA is
4    TEA_Delta : Word := 16#9E3779B9#;
5    TEA_Sum   : Word := 16#00000000#;
6
7    procedure DXPL_Process
8       (Message : in out Word_Array_2;
9        Key     : in Word_Array_4 := (others => 16#0#)) is
10   begin
11      TEA_Sum := TEA_Sum + TEA_Delta;
12      Message (0) := Message (0) +
13                     ((Shift_Left (Message (1), 4) + Key (0)) xor
14                      (Message (1) + TEA_Sum) xor
15                      (Shift_Right (Message (1), 5) + Key (1)));
16      Message (1) := Message (1) +
17                     ((Shift_Left (Message (0), 4) + Key (2)) xor
18                      (Message (0) + TEA_Sum) xor
19                      (Shift_Right (Message (0), 5) + Key (3)));
20   end DXPL_Process;
21
22 begin
23  Configuration
24     (DXPL_ALGORITHM   => "Tiny Encryption Algorithm (TEA)",
25      DXPL_ROUNDS      => 32,
26      DXPL_TERMINATION => 256);
27
28  Test_Vector
29     (DXPL_MESSAGE => (0 => 16#01234567#, 1 => 16#89abcdef#),
30      DXPL_KEY     => (0 => 16#00112233#, 1 => 16#44556677#,
31                       2 => 16#8899aabb#, 3 => 16#ccddeeff#),
32      DXPL_DIGEST  => (0 => 16#126c6b92#, 1 => 16#c0653a3e#));
33 end TEA;
```

As you can see, there is only one main procedure named `DXPL_Process`, which deals with the encryption of an input, here `Message`. TEA normally iterates the statements declared in `DXPL_Process` 32 times to perform an encryption. To integrate such iteration, you have to indicate this information as *rounds*. The procedure call to `Configuration` in line 23 takes this actually as a parameter.

In order to verify your definition of TEA, you have to add at least one test vector (see line 29).

The description of TEA above covers everything to incorporate the primitive into DARXplorer.

5. TERMINOLOGY

You've just learned, that statements of exactly one iteration of a cryptographic primitive have to be defined in `DXPL_Process`. It is one of three key procedures and is obligatory. Procedures, that DARXplorer analyzes are:

`DXPL_Process`: You have to implement the round function of a cryptographic primitive here. It will be called once in each round. It is mandatory.

`DXPL_Initialize`: If you have to assign initial values or set the key, here you go. The declaration of this procedure is optional. It will be called once before any round is processed.

`DXPL_Finalize`: The same goes for this procedure, but will be called only once after all rounds are processed.

All three procedures have the following limitations in common: It is allowed to access functions, procedures and variables globally defined. While declaring the declarative part of the procedure, only basic declarations like number or type declarations are allowed. Declaring subprograms, generic declarations et cetera is forbidden.

Statements are limited, as well. You are allowed to apply `declare` declarations and any kind of mathematical expressions. Declaring loops is instead restricted to `for` loops. Stating ranges is only allowed by means of integers. To capture the whole syntax of a valid DARXplorer-input specification, see the attached appendix.

Functions, that are declared outside of these three procedures are not analyzed. However, each additional procedure is analyzed the same way as the above named procedures and underlies the same limitations.

After declaring all subprograms and variables you need for your primitive, the `begin` declaration is mandatory. Here, you have to call the procedure `Configuration` and append at least one test vector by invoking `Test_Vector`. Both procedures are defined in the package `DXPL_Types_32` resp. `DXPL_Types_64`.

`Configuration`: Parameters are a user-defined description of the primitive, the number of rounds/iterations, `DXPL_Process` should be called and a value specifying the termination of an analyzation. A given integer `a` will be interpreted as 2^{-a}. This procedure has to be called once.

`Test_Vector`: You are able to append as much test vectors as you like, but one is obligatory to verify the primitive. Test vectors consist of an input message, an optional key and the output digest.

Last but not least, we provide you with data types and operations in 32/64 bit, that are compatible with DARXplorer. They have to be integrated with

```
with DXPL_Types_32;
```

or

```
with DXPL_Types_64;
```

Listing 1 shows a short survey of utilizable types and operations included in these packages.

```
1  type     Word_Array     is array (Integer range <>) of Word;
2  subtype Word_Array_80 is Word_Array (0 .. 79);
3  subtype Word_Array_64 is Word_Array (0 .. 63);
4  subtype Word_Array_32 is Word_Array (0 .. 31);
5  subtype Word_Array_16 is Word_Array (0 .. 15);
6  subtype Word_Array_8  is Word_Array (0 ..  7);
7  subtype Word_Array_4  is Word_Array (0 ..  3);
8  subtype Word_Array_2  is Word_Array (0 ..  2);
9
10 Rounds : Positive;
11
12 function Rotate_Left  (Value : in Word; Amount : in Natural)
13   return Word;
14
15 function Rotate_Right (Value : in Word; Amount : in Natural)
16   return Word;
17
18 function Shift_Left  (Value : in Word; Amount : in Natural)
19  return Word;
20
21 function Shift_Right (Value : in Word; Amount : in Natural)
22   return Word;
```

Listing 1. DXPL_Types_32.ads

Variable `Rounds`, declared in line 10, can be used to represent the current round number to access for example a concrete round constant. Please keep in mind, that the first round number is set to 1 and most arrays begin with 0.

You can perform the following operations on `Word`: +, −, mod, ∗, xor, and, or and not.

The package `DXPL_Types_*` makes it possible to verify in advance, that your code is a valid Ada source. You can create both versions of the package by invoking the shell script `create_types.sh` in the root of the distribution folder, which takes the path of your primitive sources, e.g. `my_scripts` as a parameter:

```
./create_types.sh my_scripts
```

You are encouraged to check, that your file is a valid Ada file.

6. INCORPORATING BLAKE-32 INTO DARXPLORER

Let's start with transferring the reference implementation of `BLAKE-32` in C into a DARXplorer-valid source, that can be analyzed.

We know, that BLAKE-32 iterates ten times and uses 32 bit types. We start with a template according to listing 2.

```
1  with DXPL_Types_32;  use DXPL_Types_32;
2
3  procedure BLAKE_32 is
4
5  begin
6    Configuration
7      (DXPL_ALGORITHM    => "BLAKE-32",
8       DXPL_ROUNDS       => 10,
9       DXPL_TERMINATION  => 256);
10
11 end BLAKE_32;
```

Listing 2. DARXplorer template

BLAKE uses a permutation, which is represented by a two-dimensional array. To access the correct permutation, constants are used. In addition, initial values have to be set. The reference C code is presented in listing 3, the corresponding Ada code in listing 4.

```
1  /*
2    the 10 permutations of {0,...15}
3  */
4  static const unsigned char sigma[][16] = {
5   {  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15 },
6   { 14, 10,  4,  8,  9, 15, 13,  6,  1, 12,  0,  2, 11,  7,  5,  3 },
7   { 11,  8, 12,  0,  5,  2, 15, 13, 10, 14,  3,  6,  7,  1,  9,  4 },
8   {  7,  9,  3,  1, 13, 12, 11, 14,  2,  6,  5, 10,  4,  0, 15,  8 },
9   {  9,  0,  5,  7,  2,  4, 10, 15, 14,  1, 11, 12,  6,  8,  3, 13 },
10  {  2, 12,  6, 10,  0, 11,  8,  3,  4, 13,  7,  5, 15, 14,  1,  9 },
11  { 12,  5,  1, 15, 14, 13,  4, 10,  0,  7,  6,  3,  9,  2,  8, 11 },
12  { 13, 11,  7, 14, 12,  1,  3,  9,  5,  0, 15,  4,  8,  6,  2, 10 },
13  {  6, 15, 14,  9, 11,  3,  0,  8, 12,  2, 13,  7,  1,  4, 10,  5 },
14  { 10,  2,  8,  4,  7,  6,  1,  5, 15, 11,  9, 14,  3, 12, 13 , 0 },
15  {  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15 },
16  { 14, 10,  4,  8,  9, 15, 13,  6,  1, 12,  0,  2, 11,  7,  5,  3 },
17  { 11,  8, 12,  0,  5,  2, 15, 13, 10, 14,  3,  6,  7,  1,  9,  4 },
18  {  7,  9,  3,  1, 13, 12, 11, 14,  2,  6,  5, 10,  4,  0, 15,  8 },
19  {  9,  0,  5,  7,  2,  4, 10, 15, 14,  1, 11, 12,  6,  8,  3, 13 },
20  {  2, 12,  6, 10,  0, 11,  8,  3,  4, 13,  7,  5, 15, 14,  1,  9 },
21  { 12,  5,  1, 15, 14, 13,  4, 10,  0,  7,  6,  3,  9,  2,  8, 11 },
22  { 13, 11,  7, 14, 12,  1,  3,  9,  5,  0, 15,  4,  8,  6,  2, 10 },
23  {  6, 15, 14,  9, 11,  3,  0,  8, 12,  2, 13,  7,  1,  4, 10,  5 },
24  { 10,  2,  8,  4,  7,  6,  1,  5, 15, 11,  9, 14,  3, 12, 13 , 0 }
25 };
26
27 /*
28   constants for BLAKE-32 and BLAKE-28
29 */
30 static const u32 c32[16] = {
```

```
31      0x243F6A88, 0x85A308D3,
32      0x13198A2E, 0x03707344,
33      0xA4093822, 0x299F31D0,
34      0x082EFA98, 0xEC4E6C89,
35      0x452821E6, 0x38D01377,
36      0xBE5466CF, 0x34E90C6C,
37      0xC0AC29B7, 0xC97C50DD,
38      0x3F84D5B5, 0xB5470917
39  };
40
41  /*
42    initial values ( IVx for BLAKE-x)
43  */
44  static const u32 IV32[8]={
45    0x6A09E667, 0xBB67AE85,
46    0x3C6EF372, 0xA54FF53A,
47    0x510E527F, 0x9B05688C,
48    0x1F83D9AB, 0x5BE0CD19
49  };
```

Listing 3. Declaration of permutation, constants and initial values of BLAKE-32

```
1   with DXPL_Types_32;  use DXPL_Types_32;
2
3   procedure BLAKE_32 is
4     ---------------------------------------
5     --  The 10 permutations of (0,..,15}  --
6     ---------------------------------------
7     subtype Permute_Positive is Natural range 0 .. 15;
8     type TwoD_Array is array (0 .. 19, 0 .. 15) of Permute_Positive;
9     SIGMA : TwoD_Array :=
10    (( 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15),
11     (14, 10,  4,  8,  9, 15, 13,  6,  1, 12,  0,  2, 11,  7,  5,  3),
12     (11,  8, 12,  0,  5,  2, 15, 13, 10, 14,  3,  6,  7,  1,  9,  4),
13     ( 7,  9,  3,  1, 13, 12, 11, 14,  2,  6,  5, 10,  4,  0, 15,  8),
14     ( 9,  0,  5,  7,  2,  4, 10, 15, 14,  1, 11, 12,  6,  8,  3, 13),
15     ( 2, 12,  6, 10,  0, 11,  8,  3,  4, 13,  7,  5, 15, 14,  1,  9),
16     (12,  5,  1, 15, 14, 13,  4, 10,  0,  7,  6,  3,  9,  2,  8, 11),
17     (13, 11,  7, 14, 12,  1,  3,  9,  5,  0, 15,  4,  8,  6,  2, 10),
18     ( 6, 15, 14,  9, 11,  3,  0,  8, 12,  2, 13,  7,  1,  4, 10,  5),
19     (10,  2,  8,  4,  7,  6,  1,  5, 15, 11,  9, 14,  3, 12, 13 , 0),
20     ( 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15),
21     (14, 10,  4,  8,  9, 15, 13,  6,  1, 12,  0,  2, 11,  7,  5,  3),
22     (11,  8, 12,  0,  5,  2, 15, 13, 10, 14,  3,  6,  7,  1,  9,  4),
23     ( 7,  9,  3,  1, 13, 12, 11, 14,  2,  6,  5, 10,  4,  0, 15,  8),
24     ( 9,  0,  5,  7,  2,  4, 10, 15, 14,  1, 11, 12,  6,  8,  3, 13),
25     ( 2, 12,  6, 10,  0, 11,  8,  3,  4, 13,  7,  5, 15, 14,  1,  9),
26     (12,  5,  1, 15, 14, 13,  4, 10,  0,  7,  6,  3,  9,  2,  8, 11),
27     (13, 11,  7, 14, 12,  1,  3,  9,  5,  0, 15,  4,  8,  6,  2, 10),
28     ( 6, 15, 14,  9, 11,  3,  0,  8, 12,  2, 13,  7,  1,  4, 10,  5),
29     (10,  2,  8,  4,  7,  6,  1,  5, 15, 11,  9, 14,  3, 12, 13 , 0)
30    );
31
32     ---------------------------------------
33     --  Constants for BLAKE-32 and BLAKE-28  --
```

```
34      ------------------------------------------------
35      c32 : Word_Array_16 :=
36        (16#243F6A88#,
37         16#85A308D3#,
38         16#13198A2E#,
39         16#03707344#,
40         16#A4093822#,
41         16#299F31D0#,
42         16#082EFA98#,
43         16#EC4E6C89#,
44         16#452821E6#,
45         16#38D01377#,
46         16#BE5466CF#,
47         16#34E90C6C#,
48         16#C0AC29B7#,
49         16#C97C50DD#,
50         16#3F84D5B5#,
51         16#B5470917#);
52
53      ------------------------------------------------
54      --  Initial values ( IV32for BLAKE-32)  --
55      ------------------------------------------------
56      IV32 : Word_Array_8 :=
57        (16#6A09E667#,
58         16#BB67AE85#,
59         16#3C6EF372#,
60         16#A54FF53A#,
61         16#510E527F#,
62         16#9B05688C#,
63         16#1F83D9AB#,
64         16#5BE0CD19#);
65
66    begin
67      Configuration
68        (DXPL_ALGORITHM    => "BLAKE-32",
69         DXPL_ROUNDS       => 10,
70         DXPL_TERMINATION  => 256);
71
72    end BLAKE_32;
```

Listing 4. Integrating permutation, constants and initial values to Ada

Next up on the agenda is handling the ten iterations of BLAKE-32. In each iteration, there are several calls to a procedure named `G32`. It would be a good idea to convert it to Ada, first. Both coming listings 5 and 6 show the transformation, whereby the latter includes all former conversions.

```
1  u32 v[16];
2  u32 m[16];
3  int round;
4
5  #define ROT32(x,n) (((x)<<(32-n))|( (x)>>(n)))
6  #define ADD32(x,y)    ((u32)((x) + (y)))
7  #define XOR32(x,y)     ((u32)((x) ^ (y)))
8
9  #define G32(a,b,c,d,i)\
10   do { \
11     v[a] = ADD32(v[a],v[b])+XOR32(m[sigma[round][2*i]], c32[sigma[ →
         ↪ round][2*i+1]]);\
12     v[d] = ROT32(XOR32(v[d],v[a]),16);\
13     v[c] = ADD32(v[c],v[d]);\
14     v[b] = ROT32(XOR32(v[b],v[c]),12);\
15     v[a] = ADD32(v[a],v[b])+XOR32(m[sigma[round][2*i+1]], c32[sigma[ →
         ↪ round][2*i]]);\
16     v[d] = ROT32(XOR32(v[d],v[a]), 8);\
17     v[c] = ADD32(v[c],v[d]);\
18     v[b] = ROT32(XOR32(v[b],v[c]), 7);\
19   } while (0)
```

Listing 5. Declaration of G32 in C

```
1  with DXPL_Types_32;  use DXPL_Types_32;
2
3  procedure BLAKE_32 is
4    ---------------------------------------
5    --  The 10 permutations of (0,..,15}  --
6    ---------------------------------------
7    subtype Permute_Positive is Natural range 0 .. 15;
8    type TwoD_Array is array (0 .. 19, 0 .. 15) of Permute_Positive;
9    SIGMA : TwoD_Array := (..);
10
11   ---------------------------------------------
12   --  Constants for BLAKE-32 and BLAKE-28  --
13   ---------------------------------------------
14   c32 : Word_Array_16 := (..);
15
16   ---------------------------------------------
17   --  Initial values ( IV32for BLAKE-32)  --
18   ---------------------------------------------
19   IV32 : Word_Array_8 := (..);
20
21   ------------
22   --  G32  --
23   ------------
24
25   M : Word_Array_16;
26
27   procedure G32
28     (A, B, C, D, I : in Integer;
29      Message       : in out Word_Array_16) is
30   begin
31     Message(A) := (Message(A) + Message(B)) +
```

```
32                    (M(SIGMA(Rounds-1, 2*I)) xor
33                    c32 (SIGMA (Rounds-1, 2*I+1)));
34      Message(D) := Rotate_Right (Message(D) xor Message(A),16);
35      Message(C) := Message(C) + Message(D);
36      Message(B) := Rotate_Right (Message(B) xor Message(C),12);
37      Message(A) := (Message(A) + Message(B)) +
38                    (M(SIGMA(Rounds-1, 2*I+1)) xor
39                    c32 (SIGMA (Rounds-1, 2*I)));
40      Message(D) := Rotate_Right (Message(D) xor Message(A), 8);
41      Message(C) := (Message(C) + Message(D));
42      Message(B) := Rotate_Right (Message(B) xor Message(C), 7);
43    end G32;
44
45  begin
46    Configuration
47       (DXPL_ALGORITHM    => "BLAKE-32",
48        DXPL_ROUNDS       => 10,
49        DXPL_TERMINATION  => 256);
50
51  end BLAKE_32;
```

Listing 6. Incorporating G32 to Ada, whereby the permutation, the constants and the initial values are omitted for readability

According to listing 6, we were required to replace the global variable `u32 v[16]` with `Message`. `Message` serves as a variable, which keeps always the current state of transformation. You have to assign your initial value to `Message`. Further on, you have to assign changes in your primitive always to it. This is currently a minor drawback in the design of DARXplorer.

As you can see in listing 6, line 32, we use the variable `Rounds` to refer to the current round number. You can consider `Rounds` as the iteration index beginning by 1. The variable `M` is placed to keep the initial input message. It will be initialized in `DXPL_Initialize`, as listing 9 illustrates. We omitted the *salt*, see lines 18 till 22, and removed the special treatment of the last block, see line 27 and 32 till 36.

```
1     ------------------------------------
2     --  Initialization of BLAKE-32  --
3     ------------------------------------
4
5     procedure DXPL_Initialize (Message : in out Word_Array_16) is
6     begin
7       --  store the original message
8       M(0 .. 15) := Message (0 .. 15);
9
10      Message( 0) := IV32(0);
11      Message( 1) := IV32(1);
12      Message( 2) := IV32(2);
13      Message( 3) := IV32(3);
14      Message( 4) := IV32(4);
15      Message( 5) := IV32(5);
16      Message( 6) := IV32(6);
17      Message( 7) := IV32(7);
18      -- removed the salt
19      --  Message( 8) := salt32 (0) xor c32(0);
```

```
20       -- Message( 9) := salt32 (1) xor c32(1);
21       -- Message(10) := salt32 (2) xor c32(2);
22       -- Message(11) := salt32 (3) xor c32(3);
23       Message( 8) := c32(0);
24       Message( 9) := c32(1);
25       Message(10) := c32(2);
26       Message(11) := c32(3);
27       -- if special case t=0 for the last block
28       Message(12) := c32(4);
29       Message(13) := c32(5);
30       Message(14) := c32(6);
31       Message(15) := c32(7);
32       -- else
33       -- Message(12) := t32(0) xor c32(4);
34       -- Message(13) := t32(0) xor c32(5);
35       -- Message(14) := t32(1) xor c32(6);
36       -- Message(15) := t32(1) xor c32(7);
37    end DXPL_Initialize;
```

Listing 7. DXPL_Initialize assigns the original message to the variable M and assigns the initial value to Message. For simplicity and readability, we omitted the rest of the template.

After declaring the permutation, round constants, the initial value and the *G32* procedure, we defined `DXPL_Initialize`. We are now ready to describe the statements for one iteration of BLAKE-32 in `DXPL_Process`. As ever, listing 8 shows the reference implementation in C and the following listing 0**??** refers to the Ada conversion.

```
1   /*  do 10 rounds */
2   for(round=0; round < 10; ++round) {
3     /* column step */
4     G32( 0, 4, 8,12, 0);
5     G32( 1, 5, 9,13, 1);
6     G32( 2, 6,10,14, 2);
7     G32( 3, 7,11,15, 3);
8
9     /* diagonal step */
10    G32( 0, 5,10,15, 4);
11    G32( 1, 6,11,12, 5);
12    G32( 2, 7, 8,13, 6);
13    G32( 3, 4, 9,14, 7);
14  }
```

Listing 8. Declaration of ten rounds of BLAKE-32 in C

```
1   ---------------------------------------
2   --  Process one round of BLAKE-32  --
3   ---------------------------------------
4
5   procedure DXPL_Process (Message : in out Word_Array_16) is
6   begin
7     --  column step
8     G32 (Message, 0, 4,  8, 12, 0);
9     G32 (Message, 1, 5,  9, 13, 1);
10    G32 (Message, 2, 6, 10, 14, 2);
11    G32 (Message, 3, 7, 11, 15, 3);
12
13    --  diagonal step
14    G32 (Message, 0, 5, 10, 15, 4);
15    G32 (Message, 1, 6, 11, 12, 5);
16    G32 (Message, 2, 7,  8, 13, 6);
17    G32 (Message, 3, 4,  9, 14, 7);
18  end DXPL_Process;
```

Listing 9. Declaring one round of BLAKE in Ada. For simplicity and readability, we omitted the rest of the template.

As you can see, there is nothing special about declaring ten rounds of BLAKE-32. Please keep in mind, that `DXPL_Process` only represents one round of BLAKE. The `for` loop is omitted, but we conserved the value of 10 in our `Configuration` (see listing 2).

Finally, we could have declared `DXPL_Finalize`, but we do not need it. Because all changes go directly into the variable `Message`, it represents automatically the digest after ten rounds.

For verification, we need to append at least one test vector. A valid test vector is declared in listing 10. The complete conversion of BLAKE-32 can be found in the folder `examples`.

```
1   --  Input Message: "abc"
2   Test_Vector
3      (DXPL_MESSAGE => (0  => 16#64636261#, 1  => 16#00000000#,
4                        2  => 16#00000000#, 3  => 16#00000000#,
5                        4  => 16#00000000#, 5  => 16#00000000#,
6                        6  => 16#00000000#, 7  => 16#00000000#,
7                        8  => 16#00000000#, 9  => 16#00000000#,
8                        10 => 16#00000000#, 11 => 16#00000000#,
9                        12 => 16#00000000#, 13 => 16#00000000#,
10                       14 => 16#00000018#, 15 => 16#00000000#),
11       DXPL_DIGEST  => (0  => 16#718cf24e#, 1  => 16#7a7d593f#,
12                       2  => 16#a88f9c24#, 3  => 16#7f7311de#,
13                       4  => 16#f5502ffa#, 5  => 16#c28a5324#,
14                       6  => 16#50c801bc#, 7  => 16#81a885ff#,
15                       8  => 16#6507bf2a#, 9  => 16#ae8fd04d#,
16                       10 => 16#a8e70a9b#, 11 => 16#5d2fc5d6#,
17                       12 => 16#4b2f6a63#, 13 => 16#72778e0f#,
18                       14 => 16#93ec20cf#, 15 => 16#152ed8cc#));
```

Listing 10. Showing a test vector declaration in Ada

7. USAGE AND VALIDATION

Subsequent to the creation of a new Ada source file, you have to type

```
./build.sh $PATH_TO_YOUR_SCRIPT
```

Don't miss to replace the placeholder $PATH_TO_YOUR_SCRIPT with the actual path to your script including the filename. If the shell script isn't running, give it the following rights and try again

```
chmod u+x build.sh
```

The framework analyses your description. If it is valid and applicable to DARXplorer, it will be integrated into the framework and automatically validated with the aid of test vectors. Remember, that you have to provide test vectors by yourself.
The output of the shell script might look like this

```
$./darxplorer.sh examples/threefish256.adb
++ The framework will be compiled ..  FINISHED
++ DARXplorer will be compiled by means of your description
   ..  FINISHED
DARXplorer has validated your description.
The application 'darxplorer' is ready to run.
```

If something went wrong, DARXplorer will tell you. The newly created DARXplorer executable allows you to analyze the described primitive, finally. If you execute `darxplorer` without any arguments, the framework validates your description on the basis of provided test vectors, again.

8. ANALYZATION

You've learned about several techniques to analyze cryptographic primitives based on a mixture of addition, rotation and XOR in chapter 2. Now, after you've successfully validated your primitive description, you can analyze it. By typing

```
./darxplorer --usage
```

usage information is print to the command line. You are able to log results and determine, which technique you want to use.

Analyzation only takes place, if you marked regions in your primitive description for analyzation. You have to implement this feature by means of annotations. We provide you for one thing `BEGIN` to start the analyzation and for another thing `END` to stop it. Both annotations have to be inserted as a Ada comment (`--`) followed by a rhombus. They are allowed only in procedures. Usage according to listing 11 and 12.

```
1   ------------------------------------
2   --  Process one round of BLAKE-32  --
3   ------------------------------------
4
5   procedure DXPL_Process (Message : in out Word_Array_16) is
6   begin
7     --  column step
8     --# BEGIN
9     G32 (Message, 0, 4,  8, 12, 0);
10    G32 (Message, 1, 5,  9, 13, 1);
11    G32 (Message, 2, 6, 10, 14, 2);
12    G32 (Message, 3, 7, 11, 15, 3);
13    --# END
14
15    --  diagonal step
16    G32 (Message, 0, 5, 10, 15, 4);
17    G32 (Message, 1, 6, 11, 12, 5);
18    G32 (Message, 2, 7,  8, 13, 6);
19    G32 (Message, 3, 4,  9, 14, 7);
20  end DXPL_Process;
```

Listing 11. Annotations for analyzation encapsulate the columnn step of BLAKE-32. Please keep in mind, that annotations work only within the defined procedure. Actually, using annotations like this has no effects.

```
1
2    M : Word_Array_16;
3
4   procedure G32
5     (A, B, C, D, I : in Integer;
6      Message       : in out Word_Array_16) is
7   begin
8     --# BEGIN
9     Message(A) := (Message(A) + Message(B)) +
10                  (M(SIGMA(Rounds-1, 2*I)) xor
11                  c32 (SIGMA (Rounds-1, 2*I+1)));
12    Message(D) := Rotate_Right (Message(D) xor Message(A),16);
13    Message(C) := Message(C) + Message(D);
14    Message(B) := Rotate_Right (Message(B) xor Message(C),12);
15    Message(A) := (Message(A) + Message(B)) +
16                  (M(SIGMA(Rounds-1, 2*I+1)) xor
17                  c32 (SIGMA (Rounds-1, 2*I)));
18    Message(D) := Rotate_Right (Message(D) xor Message(A), 8);
19    Message(C) := (Message(C) + Message(D));
20    Message(B) := Rotate_Right (Message(B) xor Message(C), 7);
21  end G32;
22    --# END
23   end G32;
```

Listing 12. Correct usage of annotations to analyze a global procedure. Minor drawback is, that you cannot analyze the column steps without analyzing the diagonal steps, too.

9. EVALUATION

If you chosed to generate log files, you will find them after processing in the folder `DARX_Logs`. For each primitive, a subfolder will be created containing the results for the chosen technique. Each log file is per default a *CSV* file. Log files are numbered ascending. The highest number contains the best results. An example can be viewed in listing 13.

```
1  --!  Copyright 2009;
2  --!    Bauhaus-University Weimar, Germany;
3  --!    Chair of Media Security / Stefan Lucks;
4  --!;
5  --!  This file was automatically created due;
6  --!  analyzation with DARXplorer 2.0. It con-;
7  --!  tains information about each stage of;
8  --!  processing like the current round, stage,;
9  --!  global and local probabilities and differentials.;
10 --!;
11 --!  DARXplorer 2.0 - Log File;
12 --!;
13 --!  Created on:  9-22-2009 16:15:15;
14 --!;
15 --!  Information about the hash function;
16 --!  Name: Threefish 256;
17 --!  Technique: Lazy Laura;
18 --!  # of Rounds: 9;
19 --!;
20 --!  Initial Differential in Round 1;
21 --!    0000_0000_0000_0008;
22 --!    0000_0000_0000_0000;
23 --!    0000_0000_0000_0000;
24 --!    0000_0000_0000_0000;
25
26
27 ;Round; Stage; Global Prob.; Local Prob.; Ham. W.; Differentials;
28
29
30 ; 1; 1;      ;            5.0000E-01  [2^(-1)]; 1;
31 ;;;;;;0000_0000_0000_0008;
32 ;;;;;;0000_0000_0000_0008;
33 ;;;;;;0000_0000_0000_0000;
34 ;;;;;;0000_0000_0000_0000;
35
36
37 ; 1; 2;      ;            1.0000E+00  [2^(-0)]; 2;
38 ;;;;;;0000_0000_0000_0008;
39 ;;;;;;0000_0000_0000_0008;
40 ;;;;;;0000_0000_0000_0008;
41 ;;;;;;0000_0000_0000_0000;
```

Listing 13. Excerpt from the log file for Threefish 256 in CSV file format, analyzed by Lazy Laura. For each stage, i.e. for each executed statement, is printed the current differential and the corresponding probability.

We log the state after each statement is processed. Due to the drawback, that the variable `Message` has to keep the current state at any time, only changes to `Message` are logged associated with a round number, hamming weight and a probability.

If you wish to write your own output format, please consult the developer documentation.

REFERENCES

Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., and Walker, J. 2008. The Skein Hash Function Family. http://www.schneier.com/skein.pdf.

Paul, S. and Preneel, B. 2005. Solving Systems of Differential Equations of Addition. http://www.cosic.esat.kuleuven.be/publications/article-566.pdf.

Whiting, D., Schneier, B., Lucks, S., and Muller, F. 2003. Helix: Fast encryption and authentication in a single cryptographic primitive. http://www.macfergus.com/helix/helix.pdf.

Whiting, D., Schneier, B., Lucks, S., and Muller, F. 2005. Phelix-Fast Encryption and Authentication in a Single Cryptographic Primitive. http://www.schneier.com/paper-phelix.html.

Wu, H. and Preneel, B. 2006. Differential Attacks against Phelix. http://www.ecrypt.eu.org/stream/papersdir/2006/056.pdf.

APPENDIX

Cryptographic primitives are written in Ada, but we limit the powerfulness of the programming language to simplify our parser. At the moment, we support the following syntax (shown in EBNF). No guarantee of completeness.

```
main_program ::=
  { <with-declaration> [ <use-declaration> ] } <package-declaration>
  { <procedure-declaration> | <function_declaration>  |
    <variables-declaration> } begin { <begin-declaration> } end ";"

with_declaration ::=
  with <identifier> { "." <identifier> } ";"

numeric ::=
  { "0" .. "9" }

identifier ::= (simplified)
  "A" .. "Z" | "a" .. "z" | "_" | <numeric>

use-declaration ::=
  use <identifier> { "." <identifier> } ";"

package_declaration ::=
  package body <identifier> is

procedure_declaration ::=
  procedure <identifier> is | procedure <identifier>
  "(" <procedure_parameter_declaration> ")" is
  begin <statements> end [ <identifier> ] ";"

proc_parameter ::=
  <identifier> { "," <identifier } ":" [ in | out | in out ]
  <identifier> [ := <expression> ]

procedure_parameter_declaration ::=
  proc_parameter { ";" proc_parameter }

expr_element ::=
  <simple_expression> [ [ "=" | "/=" | "<" | "<=" | ">" | ">=" ]
  <simple_expression> ] [ [ and | or | xor ] <simple_expression> ]

primary ::=
  "(" <expression> ")" |
  <identifier> [ "(" <expression> { "," <expression> } ")" ]

simple_expression_support ::=
  <primary> [ "**" <primary> ] [ "*" | mod | "/" ] [ "+" | "-" ]

simple_expression ::=
  [ "+" | "-" | "not" ] <simple_expression_support>
  { <simple_expression_support> }

expression ::=
  <expr_element> { <expr_element> }
```

```
statements ::=
  null ";" | return <expression> ";" | <for_loop_declaration> ";" |
  <identifier> ":=" <expression> ";" | <declare_block_declaration>
   ";" | <identifier> "(" <expression> { "," <expression> ")" ";"

for_loop_declaration ::=
  for <identifier> <numeric> ".." <numeric> loop <statements>
  end loop ";"

decl_item ::=
  <numeric> "=>" <numeric>

decl_element ::=
  <numeric> | <identifier> | "(" ( others "=>" ( <numeric> |
  <identifier> )) | <decl_item> { "," <decl_item> } ")"

variables_declaration ::=
  <identifier> { "," <identifier> } ":" [ constant ] <identifier>
  [ ":=" <decl_element> ] ";"

declare_block_declaration ::=
  declare <variables_declaration> begin <statements> end ";"

function_declaration ::=
  function <identifier> is | function <identifier>
  "(" <function_parameter_declaration> ")"
  return <identifier>  is begin <statements> end [ <identifier> ]";"

func_parameter ::=
  <identifier> { "," <identifier } ":" [ in ] <identifier>
  [ := <expression> ]

function_parameter_declaration ::=
  <func_parameter> { ";" <func_parameter> }

var_item ::=
  <numeric> "=>" <numeric>

var_element ::=
  <numeric> | <identifier> | "(" ( others "=>" ( <numeric> |
  <identifier> )) | <var_item> { "," <var_item> } ")"

variables_declaration ::=
  <identifier> { "," <identifier> } ":" [ constant ] <identifier>
  [ ":=" <var_element> ] ";"

begin_declaration ::=
  begin ( <configuration_declaration>, { test_vector_declaration } )
  end [ <identifier> ] ";"

configuration_declaration ::=
  [ <identifier> "." ] <identifier> := "("
  [ DXPL_ALGORITHM => ] <identifier> "." <identifier> ","
  [ DXPL_ROUNDS    => ] { <numeric> } ")" ","
  [ DXPL_TERMINATION => ] { <numeric> } ")" ";"
```

```
test_vector_declaration ::=
  [ <identifier> "." ] <identifier> := "("
  [ DXPL_Message =>  ] <array_declaration> ","
  [ [ DXPL_Key   =>  ] <array_declaration> "," ]
  [ DXPL_DIGEST  =>  ] <array_declaration> ")" ";"

array_element ::=
  [ <numeric>  => ] <numeric>

array_declaration ::=
  "(" array_element { "," array_element } ")"
```