DARXplorer 2.0 – Developer Documentation

DENNIS HOPPE
Bauhaus-University Weimar

DARXplorer is a toolbox for cryptanalysts and cipher designers. It 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 (1) aiding the designers of new primitives to defend against attacks and (2) analyzing existing primitives to find attacks.

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* [Lucks 2008b] used in the hash function *Skein* [Ferguson et al. 2008]. Threefish based on a mixture of addition modulo 2^{64}, rotation and xor. 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. Due to the fact, that the framework was completely rewritten, this developer documentation will give you detailed information about the implementation for further development.

For more information regarding DARXplorer in general, see Lucks [2008a].

This developer documentation is about installation and usage of DARXplorer in chapters 2 and 3. Chapter 4 presents the object structure in detail. It is shown, which steps are needed to integrate an input source file into DARXplorer. Chapter 5 attends the supported syntax for writing declarations of cryptographic primitives. Subsequent to this chapter discusses chapter 6 all implemented techniques in detail and shows how to add new techniques. Chapter 7 deals with user-defined formatting of log files, before in chapter 8 challenges and future work are discussed.

2. PREREQUISITES

All you need is *GNAT*, the free-software compiler for Ada in version 4.3 or higher. If you are running an UNIX based operating system like Ubuntu, you can get the current GNAT compiler simply by typing in your preferred terminal:

```
$ sudo apt-get install gnats
$ gnatmake -version
```

You should see the following output:

```
GNATMAKE 4.3.4
```

```
Copyright (C) 1995-2007, Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.
```

DARXplorer does not use any GNAT-specific features. It is likely, that DARXplorer will compile with GNAT versions prior to 4.3. We have tested the current distribution successfully under *Ubuntu* 9.04 with GNAT 4.3 installed. Further on, DARXplorer is running under *MAC OS X* (Version 10.5.x) with GNAT GPL 2009 installed. If you use MAC OS X 10.6 (Snow Leopard), you have to upgrade to an experimental branch of GNAT 4.4, because GNAT GPL 2009 is currently not running on Snow Leopard.

For the development of DARXplorer are no special tools required.

3. INSTALLATION

After the successful installation of GNAT you are ready to download DARXplorer. Checkout the sources from the repository to your local folder by typing

```
$ svn co svn://medsec.medien.uni-weimar.de/darx
```

Please have your username and password on tap.

In the checkout directory, you will find a `Makefile` and a shell script named `build.sh`. For development purposes, the Makefile is sufficient. The shell script offers a convenient way for end users to use DARXplorer.

3.1 Makefile

The `Makefile` offers you the following targets:

(1) `compiler` Compiles the DARXplorer framework and creates the executable `dxpl-compiler-cli`, that can be used to convert primitive declarations in Ada. While transforming the description, several files will be added to the framework.

(2) `run-compiler` Usually, if you went through step (1), you have to call the executable with your desired source file. The target `run-compiler` incorporates both steps by taking the script as an argument, e.g. : `make run-compiler ARG=examples/threefish256.adb`. This target returns `TRUE`, if no errors occurred.

(3) `function` Additional files created in step (1) or (2) have to be compiled a second time. This target generates the `darxplorer` executable, finally.

(4) `clean` Cleans up the object directory.

All in all, if you want to create a DARXplorer instance incorporating Threefish 256, you have to execute the following commands in the root of the distribution:

```
$ make run-compiler ARG=examples/Threefish256.adb
$ make function
```

3.2 Shell script

The shell script applies the same tasks as the `Makefile`, but provides a simplified interface to allow end users to use DARXplorer in a more convenient way. Only one call to `build.sh` is sufficient to build the final executable:

```
$ ./build.sh examples/Threefish256.adb
```

By calling the shell script without arguments or by means of `--usage` as a parameter, it will display some useful help on the command line.

4. PROJECT STRUCTURE

DARXplorer currently shows the following project structure, if directly checked out from the repository:

trunk
..**bc-20090226**
..`build.sh`
..`Makefile`
..**examples**
..**src**
....**compiler**
......**lexer**
......**parser**
........**support**
....**function**
..**tests**

Consequent sections will enlighten the contents of each folder.

4.1 bc-20090226

We use some data structures provided by the *Ada95 Booch Components*, especially graphs and queues. These data structures are not provided by Ada itself. Unfortunately, the implementations of Ada95 Booch needed some revision by ourselves to offer the behavior, we required. You can find the adapted priority queue in the packages

`bc-containers-queues-ordered-bounded.ads` and
`bc-containers-queues-ordered-bounded.adb`

and the aligned directed graph in the packages

`bc-graphs-directed-sorted.ads` and
`bc-graphs-directed-sorted.adb`

in the `src` folder. DARXplorer is interfacing the Ada95 Booch components by means of the `dxpl-booch-*` packages, which can be found in the source folder, as well.

The components are updated frequently by two very responsive authors. So it might be a good idea to check their repository once in a while. The interfaces in DARXplorer and the customized versions of the priority queue and the directed graph should not be touched by updates. Just replace the folder `bc-20090226` with a new version and match the path to the components in the `Makefile` plus the shell script.

4.2 examples

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 be started, if you want to learn, how to write descriptions by yourself. At present, we support 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.

4.3 src

Contains the source distribution of DARXplorer. It includes the following packages:

Packages	Description
`dxpl-io-*`	Implementing a project wide debugger, logger and formatter.
`dxpl-round_function-*`	Interfaces and basic methods of the round function.
`dxpl-support-*`	Implementation of algorithms provided by Lipmaa and Moriai to examine the differential properties of addition.
`dxpl-technique-*`	Providing implementations of techniques: Lazy Laura, Greedy Grete, Pedantic Petra and Fuzzy Fiona.
`dxpl-toolbox-*`	Analyzation framework for each technique

4.4 src/compiler

The compiler is used to integrate a description of a cryptographic primitive into the framework. Therefore, several files are automatically generated and copied to the subdirectory `src/function`.

The subsequent table lists important packages and files.

Packages/Files	Description
`dxpl_types.dummy`	Declaration of project wide data types. This file is used to generate automatically consistent Ada sources declaring data types based on 32 and 64 bit.
`dxpl-compiler-cli`	Implementation of the command line interface of DARXplorer.
`dxpl-compiler-*`	We provided a separate debug package for the compiler. Also, implementations of a symbol table and a syntax-tree can be found here.

The syntax tree is crucial in DARXplorer. It is used to parse arbitrary mathematical expressions in a primitive description. See chapter 5.

4.5 src/compiler/lexer

A very efficient implementation of a lexer is implemented. The vocabulary, the lexer and compiler are aware of, has to be defined in the corresponding package. The lexer is divided into three parts: the lexer itself, the vocabulary and a reader package to provide input/output operations. The lexer is used by the compiler to read in primitive descriptions.

4.6 src/compiler/parser

The parser is used to parse and interpret the primitive description. Implementation is comprehensive, but can be simplified by studying the *BNF* syntax. about supported statements. Naming of parser elements is aligned to the BNF syntax. To keep source files small, many parts are outsourced by means of `separate` (Ada keyword). The parser generates the following packages, if the input was `examples/threefish256.adb`. They are stored in `src/function`.

```
with Interfaces;
package DXPL is
  type Word is new Interfaces.Unsigned_64;
  subtype Rounds is Positive range 1 ..  9;
end DXPL;
```

Listing 1. dxpl.ads

```
package DXPL.Support.State is
  type Round_State is new Basic_State with
    record
      Message : Word_Array_4 := (others => 16#0#);
      key : word_array_4;
    end record;

  function Get_Name return String;
  function Get_Digest (State : in Round_State) return Word_Array;

  procedure Set_Digest (State : in out Round_State; Item : in →
      ↪ Word_Array);
end DXPL.Support.State;
```

Listing 2. dxpl-support-state.ads

```
package body DXPL.Support.State is
  procedure Set_Digest (State : in out Round_State; Item : in →
      ↪ Word_Array) is
  begin
    State.Message := Item;
  end Set_Digest;

  function Get_Digest (State : in Round_State) return Word_Array is
  begin
    return State.Message;
  end Get_Digest;

  function Get_Name return String is
  begin
    return "Threefish 256";
  end Get_Name;
end DXPL.Support.State;
```

Listing 3. dxpl-support-state.adb

```
with DXPL.Types;  use DXPL.Types;

procedure DXPL.DARXplorer is
  Threshold : Conditioned_Float := 2.0 ** (- (Integer (256)));
  Number_of_Modules : Integer := 17;
  Name : String := "Threefish 256";

  procedure Process is separate;

begin
  Process;
end DXPL.DARXplorer;
```

Listing 4. dxpl-darxplorer.adb

As you can see, listings 1 to 4 contain just a minimum of information needed to compile the framework. The architecture of DARXplorer is very generic. Up until now, we were missing the actual representation of the cryptographic primitive. The package `dxpl-round_function.adb` has its implementation.

The procedure `DXPL_Process` in the primitive description is evaluated statement per statement. Each time, the parser encounters a mathematical statement, it will build a corresponding syntax tree. The tree is used to detect additions modulo 2^{32} or 2^{64}. Each addition has to be analyzed by one of the techniques we declared in advance (see chapter 6). A problem occurred once, if more than one addition is included in the mathematical statement. Here, we have to examine each addition on its one. The syntax tree helps us to create temporary statements of the initially expression. Each sole statement is encapsulated in its own function.

Let's have a look at the main parts of the resulting `dxpl-round_function.adb`, which are self-explanatory.

```
package body DXPL.Round_Function is

  function Process_1(State : access Rnd_State) return access Word is
  begin
    Analyse (State.Message(0), State.Message(1), State.Message(0),
      State.Output_Candidates, State.Local_Probability);
    return State.Message(0)'Access;
  end;

  [..]

  function Process_17 (..) return access Word is
  begin
    [..]
  end;

  function Get_Message (Index : in Positive)
    return Word_Array is separate;
  function Get_Digest  (Index : in Positive)
    return Word_Array is separate;
  function Get_Key     (Index : in Positive)
    return Word_Array is separate;
  procedure Initialize (State : access Round_State) is separate;
  procedure Finalize (State : access Round_State) is separate;
  function  Number_of_Tests return Positive is separate;

begin
  Module_Sequence :=
      (1 => Process_1'Access, [..], 17=> Process_17'Access);
  Test_Vectors.Batch :=
      new Validation_Array'(1 => (Message => new Word_Array'([..]),
                                  Digest => new Word_Array'([..])));
end;
```

Listing 5. dxpl-round_function.adb

4.7 tests

Test cases are produced in many domains according to the in EBNF declared syntax presented in chapter 5. We provided a *Makefile* in each subdirectory. Tests are based on *tg* [Spiegel 1997], a simple test driver generator for Ada. Unfortunately, *tg* fails while processing, if more than one test raises an exception. Distinct tests with each example were correctly executed. To run a test, go into a subdirectory and type

```
make testgen
make compiler
make run
```

5. SUPPORTED SYNTAX IN DARXPLORER

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 } ")"
```

Listing 6. Supported syntax in DARXplorer

6. TECHNIQUES FOR ANALYZATION

As it turned out, the Achilles heel of these primitives was some vulnerability to differential cryptanalysis, mostly based on XOR differences. When analysing a given primitive of this type, searching for good xor differences appears to be the way to go.

The toolbox will implement four different techniques to find "good" characteristics. Usually, the differential behaviour of a low Hamming-weight difference is better predictable (i.e., with a higher probability) than the behaviour 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.

6.1 Implemented techniques

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 optimises 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 $\frac{1}{2}$.

6.2 Adding new techniques

Techniques are defined in packages named `dxpl-technique-*`. Each of the them should implement a procedure with the following parameter list:

```
X, Y : in Word;
Z    : in out Word;
Output_Candidates : in out Word_Vector.Vector;
Probability       : in out Conditioned_Float;
```

to support the integration of a particular technique in the framework. The procedure serves as a parameter to the generic round function declared in `dxpl-round_function.adb`. See procedure `Analyse` in listing 5.

Listings 7 and 8 show the default behavior, that should be implemented when declaring a new procedure to overwrite `Analyse`. Here, both terms of the sum were just added up.

```
with DXPL.Round_Function;
with DXPL.Toolbox;
with DXPL.Toolbox.Trudy;
with DXPL.Types;   use DXPL.Types;

pragma Elaborate_All (DXPL.Round_Function);
pragma Elaborate_All (DXPL.Toolbox.Trudy);

generic
  Number_of_Modules : Integer;
package DXPL.Technique.Trudy is

  procedure Solve
    (X, Y              : in     Word;
     Z                 : in out Word;
     Output_Candidates : in out Word_Vector.Vector;
     Probability       : in out Conditioned_Float);

  package Characteristic_Function is new DXPL.Round_Function
    (Analyse           => Solve,
     Number_of_Modules => Number_of_Modules);

  package DARX is new Toolbox
    (Round_Characteristic => Characteristic_Function,
     Toolbox_Name         => "Trusty Trudy",
     Toolbox_Threshold    => Toolbox_Threshold,
     Logging_Formatter    => Default_Formatter);

  package DARXplorer is new DARX.Trudy;
  Toolbox : DARXplorer.Instance;
end DXPL.Technique.Trudy;
```

Listing 7. dxpl-technique-trudy.ads

```
package body DXPL.Technique.Trudy is
  procedure Solve
    (X, Y              : in     Word;
     Z                 : in out Word;
     Output_Candidates : in out Word_Vector.Vector;
     Probability       : in out Conditioned_Float) is
    pragma Warnings (Off, Output_Candidates);
    pragma Warnings (Off, Probability);
  begin
    Z := X + Y;
  end Solve;
end DXPL.Technique.Trudy;
```

Listing 8. dxpl-technique-trudy.adb

It is just that simple. If you had to implement the behavior of *Lazy Laura*, you need to represent the addition by xor and compute the probability, see listing 9 for a solution.

```
with DXPL.Support.Differential_Addition;
use  DXPL.Support.Differential_Addition;

package body DXPL.Technique.Laura is

  function "+" (L, R : Word) return Word is
  begin
    return L xor R;
  end "+";

  procedure Compute_Optimal_Output
    (X, Y              : in     Word;
     Z                 : in out Word;
     Output_Candidates : in out Word_Vector.Vector;
     Probability       : in out Conditioned_Float) is
    pragma Warnings (Off, Output_Candidates);
  begin
    Z := X + Y;
    Differential_Probability (X, Y, Z, Probability);
  end Compute_Optimal_Output;
end DXPL.Technique.Laura;
```

Listing 9. dxpl-technique-laura.adb

7. PROVIDE A USER-DEFINED FORMAT FOR LOG FILES

Analyzation produces suitable log files. Formatting is user-defined and can be changed by subclassing `dxpl.io.formatter.ads`. Per default, a CSV format is chosen to log each stage and round, i.e.:

```
;Round; Stage; Global Probability; Local Probability; Hamming Weight; Differentials;

; 1; 1; ; 5.0000E-0 [2^(-1)]; 1;
;;;;;;0000_0000_0000_0008;
;;;;;;0000_0000_0000_0008;
;;;;;;0000_0000_0000_0000;
;;;;;;0000_0000_0000_0000;
```

Probabilities are presented in decimal with four decimal places and rounded to the power of two, which increases readability. Words are represented with leading zeros and underscores every fourth digit.

8. CHALLENGES AND FUTURE WORK

DARXplorer made a significant leap compared to its predecessor. The framework can handle Ada-based inputs to allow the declaration of arbitrary cryptographic primitives based on a mixture of addition, rotation and xor. We showed, that several primitives can be declared to be compatible with DARXplorer. Nevertheless, the syntax DARXplorer currently understands is very limited.

Producing examples for DARXplorer showed, that converting cryptographic primitives to Ada is a error prone task. Most primitives are declared in *C*. It might be helpful to replace the parser for Ada with a parser, that converts C code in Ada. In this way, primitives could be applied to DARXplorer by means of simple copy and paste.

Then again, declaration of primitives in Ada allows designers to fall back upon the powerfulness of Ada in scopes, where DARXplorer does not need to analyze the source code. These sections are variable declarations, additional functions and procedures outside from `DXPL_Initialize`, `DXPL_Finalize` and `DXPL_Process`.

A major drawback of DARXplorer 2.0 as opposed to its predecessor is, that the framework no longer supports backward analyzing. First tests showed that maximum rounds, that could be reached with the former DARXplorer by starting in the middle and perform both forward and backward analyze, has been bisected. To support backward analyzation, one can either declare a backward round of the primitive by its own or, much better, find a way to automatically reverse the forward declaration within DARXplorer itself.

Further on, it would help matters, if the framework runs distributed on several cores. At the moment, only one core is occupied with analyzation. Each analyzation involves pursue an input difference through as much rounds as possible. For each input difference, an extra thread could be created.

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`.

Lucks, S. 2008a. DARX Finder 4 Threefish.

Lucks, S. 2008b. Skein and Threefish: a Hash Function and its Internal Block Cipher.

Spiegel, A. 1997. A Simple Test Driver Generator for Ada programs – tg Version 3.1. `http://www.free-software-consulting.com/projects/tg/`.