Advanced Software Protection:
Integration, Research and Exploitation

D2.01
Early White-Box Cryptography and Data Obfuscation Report

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	November 1, 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10

Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D2.01
WP and tasks contributing:	WP 2 / Task 2.1, Task 2.2
Due date:	October 2014 – M12
Actual submission date:	31 October 2014

Responsible Organization:	FBK
Editor:	Mariano Ceccato
Dissemination level:	Public
Revision:	1.0

Abstract:
The goal of this deliverable is to report about the project activities on data protection, they are Task 2.1 *data obfuscation* and Task 2.2 *White-Box cryptography*. We present a survey of techniques for data obfuscation and for white-box cryptography along with an evaluation of their pros and cons. Based on the outcome of the analysis performed, the most promising variants are selected for implementation and integration in the ASPIRE compiler tool chain. Additionally, preliminary implementation is subject to experimental validation to measure performance overhead due to protection.

Keywords:
Data obfuscation, White-Box Cryptography

Editor
Mariano Ceccato (FBK)

Contributors (ordered according to beneficiary numbers)
Brecht Wyseur, Patrick Hachemane (NAGRA)
Mariano Ceccato, Roberto Tiella (FBK)
Jerome D'Annoville (GTO)

The ASPIRE Consortium consists of:

Ghent University (UGent)	Coordinator & Beneficiary	Belgium
Politecnico Di Torino (POLITO)	Beneficiary	Italy
Nagravision SA (NAGRA)	Beneficiary	Switzerland
Fondazione Bruno Kessler (FBK)	Beneficiary	Italy
University of East London (UEL)	Beneficiary	UK
SFNT Germany GmbH (SFNT)	Beneficiary	Germany
Gemalto SA (GTO)	Beneficiary	France

Coordinating person: Prof. Bjorn De Sutter
E-mail: coordinator@aspire-fp7.eu
Tel: +32 9 264 3367
Fax: +32 9 264 3594
Project website: www.aspire-fp7.eu

Executive Summary

This deliverable is the first report on the approaches to data protection within the ASPIRE framework. There are two major contributions of this deliverable, they are (1) results about the data obfuscation achieved in Task 2.1 and (2) results about white-box cryptography achieved in Task 2.2.
We started filling a taxonomy of the most promising data obfuscation transformations available in the state of the art. A feature grid analysis has been conducted, by listing peculiarities, strong and weak points of several obfuscations. Based on this feature evaluation, we selected four obfuscation schemes to be implemented within ASPIRE. They are xor integer encoding, residue number coding, merge scalar variable and convert static to procedural data.
To understand what are the most appropriate variables to obfuscate, we elaborated the notion of data distance. Those variables that concur to define the value of a sensitive variable are said to be *proximate* to such sensitive variable. Based on this notion of distance, we define the Data-Proximity-Graph DPG, a data structure that we use to define sets of variables to obfuscate (obfuscation configurations).
The four selected obfuscations have been implemented in TXL as source-to-source rewriting rules. Implemented obfuscations have been assessed on a case study C program, a license check routine. Obfuscated code has been compared with clear code in terms of obfuscation cost and improved security. Obfuscation cost is estimated by the impact on runtime overhead (obfuscated code takes longer to run), on memory overhead (obfuscate code requires more memory) and number of encode/decode operations occurred at runtime (obfuscated code executes additional operations). The improved security brought by obfuscation is approximated by the number of program variables that are subject to data obfuscation.
As expected, the most advanced obfuscation (i.e., residue number coding) involves higher cost than the most simple obfuscation schemes. However, memory overhead and execution time overhead grow linear with the number of obfuscated variables.
In the second part of this deliverable, we present the activities on white-box cryptography as conducted in Task 2.2. This comprises three parts as described in the ASPIRE Description of Work: (1) the development of a White-Box Tool that manages the process of generating white-box implemenations, (2) the research on new white-box schemes with strong (provable) security properties, and (3) the research and development of more practical white-box schemes such as dynamic or time-limited white-box implementations. The first two parts have received most attention in the first year of the project, the practical white-box implementation are the main activity that remains for the remainder of the Task – as planned in the ASPIRE Description of Work.
NAGRA has been contributing a framework that manages the generation process of white-box code to the consortium. This is denoted as the White-Box Tool (WBT). This tool has been further developed as an activity in Task 2.2. to support additional features that are desired for ASPIRE, such as the support for dynamic white-box implementation and generation management for supporting renewability. To ease the integration of the WBT in the ASPIRE Compiler Tool Chain (ACTC), a frontend has further been developed that presents simplified interfaces to the ACTC. This frontend is denoted as the WBT for ASPIRE (WBTA).
As a second main activity, Gemalto and NAGRA had an activity scheduled on research into new white-box schemes with provably secure properties; based on ideas in Fully Homomorphic cryptography and Multivariate cryptography respectively. The idea in the multivariate track is to rely on the multivariate decomposition hardness problem, where a multivariate cipher can be white-boxed by introducing annihilating multivariate maps into the implementation – similar to the idea of introducing annihilating encodings in the original white-box approach. This research has been conducted in three main parts: (1) investigate the existing constructions on multivariate ciphers to figure out what kind of constructions may be adequate for white-box, (2) investigate the impact on performance to figure out what kind of parameters for such constructions may be appropriate, and (3) perform a thorough security analysis to figure out what level of security can be achieved.

Our conclusion is that multivariate WBC is a viable approach, but only for ciphers with small block size for which low performance and large implementation size is acceptable.
With respect to WBC based on Fully Homomorphic Encryption, Gemalto presents the reasons for not having started their work as described in the ASPIRE Description Work, as well an alternative protection they propose to work on in ASPIRE.

Contents

List of Figures

List of Tables

1 Introduction

Section authors:
Mariano Ceccato (FBK)

Data integrity and confidentiality represent crucial assets to protect among those that are potentially threatened by man-at-the-end attacks. The objective of Task 2.1 and Task 2.2 of the ASPIRE project is therefore to elaborate effective techniques to prevent data extraction and data tampering. This deliverable presents the outcome of these two tasks to implement solutions for data obfuscation (Task 2.1) and white-box cryptography (Task 2.2), respectively in Part I and Part II.
For data obfuscation and for white-box cryptography, the elaboration of the first prototypes is based on the analysis of the state of the art. The aim of reviewing the state of the art is to identify the most appropriate candidates based on an evaluation of their advantages and disadvantages. Preliminary working prototypes are presented and empirically assessed to quantify the performance overhead due to protection.
This document is the first deliverable about data protection (data obfuscation and white-box cryptography). The presented working prototypes will be further improved and extended during the development of the project. Moreover, they will be adapted to fit in the ASPIRE compiler tool chain and integrated with the other protection strategies that will be delivered by ASPIRE.

Part I
Data Obfuscation

This part of the document is devoted to describe activities concerning data obfuscation techniques performed during the first year of the ASPIRE project. In Section 2 we present an overview of data obfuscation techniques reported in literature. From this set of transformations, we have selected four techniques to be implemented in the ACTC. The selection was based on a feature analysis devoted to evaluate pros and cons of each transformation. Results are presented in Section 3. Selected techniques were implemented in a tool that is integrated in the ACTC. The implementation is based on a pattern/rule paradigm that is explained in Section 4. Costs and benefits of applying data obfuscation to programs are discussed in Section 5. In this section the concept of variable neighbourhood is introduced as a way to control the trade off between obfuscation level and performance overhead. Implementations of chosen data obfuscation techniques were subjected to an empirical evaluation and results are presented in Section 6. Finally, Section 7 comments on other works related to data obfuscation.

2 Taxonomy of Data Obfuscation

Section authors:
Roberto Tiella (FBK)

Data obfuscation transformations change programs with the aim of hiding both variable content and usage. In Collberg et al. data obfuscation transformations are classified as depicted in Figure 1[21]. Classes written in light gray collect techniques specific to object-oriented programming languages such as Java or C++ and, for this reason, out of scope of the present document.
The first classification level distinguishes transformations in:

- **Storage & Encoding**: Transformations that change how (scalar) data are represented and stored in memory;
- **Aggregation**: Transformations that alter how data, both scalar variables and arrays, are aggregated;
- **Ordering**: Transformations that permute items in data structures as, for example, changing the order of items in an array of integers.

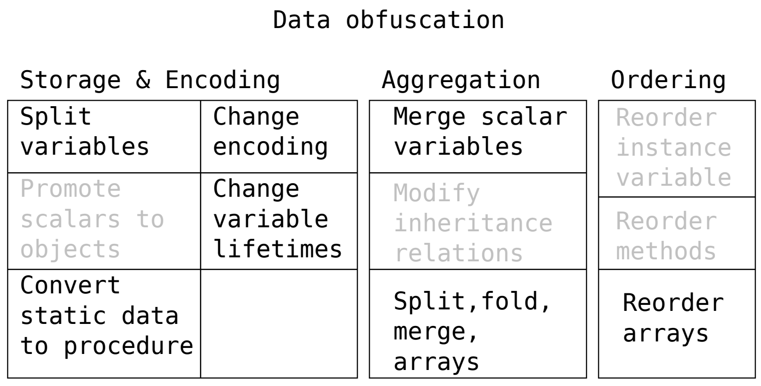

Data obfuscation
Storage & Encoding
Aggregation
Ordering
Split variables
Change encoding
Merge scalar variables
Reorder instance variable
Promote scalars to objects
Change variable lifetimes
Modify inheritance relations
Reorder methods
Convert static data to procedure
Split, fold, merge, arrays
Reorder arrays

Figure 1: Data obfuscation taxonomy presented by Collberg et al. [21]

In the following sections each transformation subclass is presented.

Notation and Definitions

In the following sections $\mathbb{Z}$ denotes the set of integers, x and y integers are said to be n-modulo equivalent (written $x \equiv_n y$) iff $\exists k \in \mathbb{Z} : x - y = kn$. $[x]_n$ indicates the class of integers that are equivalent to x with respect to the n-modulo equivalence relation and $\mathbb{Z}/n\mathbb{Z}$ the set of those classes. Furthermore $\lfloor x \rfloor$ is the largest integer not greater than x, $\mod(x, n)$ is the remainder of x divided by n and $\gcd(m, n)$ the greatest common divisor between m and n.

2.1 Storage and Encoding Data Obfuscations

As written in Collberg et al, obfuscating storage transformations attempt to choose non conventional memory layout for dynamic as well as static data [21]. Splitting a single 32 bits integer variable in four 8 bits variables is an example of storage transformation. Similarly, encoding transformations attempt to choose unnatural encodings for common data types.
Encoding a Boolean value as an integer value where any even number represents True and any odd number represents False is an example of encoding transformation for the C programming language where True is encoded as any non null integer number while 0 represents False. Storage and encoding transformations often go hand-in-hand, but they can sometimes be used in isolation.
Source-to-source data obfuscation transformations change fragments of programs such as variable declarations, assignments and variable/constant usages, written in a specific programming language. In particular, the behavior of a *change encoding* transformation is completely specified given how the transformation acts on variable values. Thus the transformation can be formally defined as a set of 1-to-1 encoding functions $\{e_v(\cdot)\}_{v \in V}$ for a given set V of variables to obfuscate. To avoid overloading the notation, where possible, variable indexes are omitted and an encoding function is denoted simply as $e(\cdot)$. Furthermore, in what follows, $d(\cdot)$ is used to denote the inverse of an encoding function $e(\cdot)$, namely $d(\cdot) = e^{-1}(\cdot)$. The C language is assumed as reference language in the rest of the document.

Base transformation:

Without any other assumptions, a transformation defined by an encoding function $e(\cdot)$ is applied to source code as follows:

- When a value v, as a results of the evaluation of an expression exp, is assigned to an encoded variable x, v must be encoded, i.e., a statement like $x = exp$; becomes $x = e(exp)$.
- When an encoded variable x is used, the decode function d has to be applied to its value, an expression like $x + exp$ becomes $d(x) + exp$.

For example, suppose that three variables a, b and c must be encoded using the same encoding function $e(\cdot)$. If variables a and b store encoded values and the encoded value of the expression $a + b$ has to be assigned to an encoded variable c then c must be assigned with the value of the expression $e(d(a) + d(b))$.

Parametric Encoding

Functions e and d can depend on a (multidimensional) parameter p, namely

$$e(\cdot) = e_p(\cdot)$$

$$d(\cdot) = d_p(\cdot),$$

where p is chosen according to a particular strategy. Examples of strategies are (a) encoding different variables using different values for p and (b) choosing a new value for p at every definition point of a variable.

Example

A simple but quite frequently used parametric encoding function involves the bitwise XOR operator. If $\oplus$ denotes the XOR Boolean operator, the function $e(\cdot)$ is defined as follows:

$$e(x) = x \oplus p$$

where p is an integer constant. From the property of the XOR operator that $(x \oplus p) \oplus p = x$, it follows that the decoding function $d = e$. This encoding is usually called *XOR masking*. Figure 2 contains snippets of code before and after having applied the XOR masking (with $p = 12$).

```
int a = 5;                        int a = 9; // 9 = 5^12
int b = 8;                        int b = 4; // 4 = 8^12
int x = a+b;            ⇒         int x = ((a^12)+(b^12))^12;
...                               ...
printf("%d\n",x);                 printf("%d\n",x^12);
```

Figure 2: A code snippet for the XOR masking example

The encoding function can be applied in a similar way to arrays as shown in Figure 3. Each character of the array must be decoded before use.

```
                                          char * message =
                                              { 'A'^p, ' '^p, 'm'^p, ...,
                                              '\0' ^ p };
char * message =                          ...
   "A message to be hidden";      ⇒       for (i=0; i<m_len; i++) {
...                                          putchar(message[i]^p);
printf("%s\n",message);                   }
                                          putchar('\n');
```

Figure 3: A code snippet that illustrate the application of the XOR operator to an array of chars

Known Uses of Storage and Encoding Change

Table 1 lists XOR masking and other encoding techniques known to be employed by practitioners, malware programmers in particular [17].

Method	Original	Encoded
XOR masking	x	$x \oplus p$
XOR masking with varying parameter	$x[i]$	$x[i] \oplus p_{mod(i,N_p)}$
ROT13	$x \in \{'A', ..., 'Z'\}$	$chr(mod(asc(x) - 65 + 13, 65) + 65)$
BASE64	$x[]$	$BASE64(x)$ as described in RFC4648 [54]

Table 1: Encoding transformations used by practitioners

2.1.1 Homomorphic Encoding

A function $f : X \to Y$ defined on a set X, provided with an operation $+$, to a set Y, provided with the operation $+'$, is called *homomorphic* if it satisfies, for all $x_1, x_2 \in X$, the following condition:

$$f(x_1 + x_2) = f(x_1) +' f(x_2). \tag{1}$$

When a homomorphic encoding function $e(\cdot)$ is used, source code obfuscating transformations can be simplified leveraging Eq. 1, holding:

$$e(d(a) + d(b)) = e(d(a)) +' e(d(b)) = a +' b.$$

The simplification avoids the need to decoding encoded values to perform the operation as in the general case presented before in Section 2.1.
As an example of homomorphic encoding, consider the following function $e : [0, \lfloor\sqrt{N}\rfloor - 1] \to \mathbb{Z}/N\mathbb{Z}$ from the set of non-negative integers less than $\lfloor\sqrt{N}\rfloor$ and the set of equivalence classes modulo N:

$$\begin{aligned} e(x) &= [x]_N \text{ represented by the element } Np + x,\ p \text{ randomly chosen} \\ d(y) &= y \bmod N \end{aligned}$$

The encoding is homomorphic both for the $+$ and the $*$ operations:

$$e(x) +' e(y) = Np_1 + x +' Np_2 + y = (p_1 + p_2)N + (x + y) \equiv_N e(x + y)$$

$$e(x) *' e(y) = (Np_1 + x) *' (Np_2 + y) = (N^2 p_1 p_2 + Np_2 x + Np_1 y) + xy \equiv_N e(x * y)$$

```
int a = 8;             int a = 108 ;
int b = 3;             int b = 103 ;
int x = a+b;     ⇒     int x = a+b;    // x = 211
...                    ...
printf("%d\n",x);      printf("%d\n",d(x)); // d(x) = d(211) = mod(211,100) = 11
```

Figure 4: A code snippet for a homomorphic encoding function

2.1.2 Residue Number Coding

Residue Number Coding (RNC) generalizes the homomorphic transformation presented in the previous example [73].

Residue Number Coding:

Having chosen $m_1, m_2, ..., m_u \in \mathbb{Z}$ so that $gcd(m_i, m_j) = 1$ if $i \neq j$, and $n = m_1 \cdot m_2 \cdot ... \cdot m_u$, $x \in [0, n-1]$ can be encoded in the following way:

$$e(x) = ([x]_{m_1}, [x]_{m_2}, ..., [x]_{m_u}).$$

The decoding function d is defined such that:

$$d([y]_{m_1}, [y]_{m_2}, ..., [y]_{m_u}) = [y],$$

where $[y]$ exists and it is unique by the "Chinese Remainder Theorem" and can be computed using Euclid's extended algorithm to compute the gcd [46].
Operations in the encoded domain are defined "per component":

$$([x_1]_{m_1}, [x_2]_{m_2}, \ldots) + ([y_1]_{m_1}, [y_2]_{m_2}, \ldots) = ([x_1 + y_1]_{m_1}, [x_2 + y_2]_{m_2}, \ldots)$$
$$([x_1]_{m_1}, [x_2]_{m_2}, \ldots) * ([y_1]_{m_1}, [y_2]_{m_2}, \ldots) = ([x_1 * y_1]_{m_1}, [x_2 * y_2]_{m_2}, \ldots)$$

Figure 5 shows an example of encoding integer variables x and y using RNC. Constants used for the encoding are $m_1 = 31$ and $m_2 = 37$, representants are chosen randomly.

```
...
   int x = 12;
   int y = 7;

   int z = x+y;
   int w = x*y;

   printf("z=%d, w=%d\n",z,w);
...
```

$\Rightarrow$

```
...
   int x1 = 1965; // 1965 % 31 = 12
   int x2 = 1973; // 1973 % 37 = 12

   int y1 = 1433; // 1433 % 31 = 7
   int y2 = 2634; // 2634 % 37 = 7

   int z1 = x1+y1;
   int z2 = x2+y2;

   int w1 = x1*y1;
   int w2 = x2*y2;

   printf("z=%d, w=%d\n",d(z1,z2),d(w1,w2));
...
```

Figure 5: An example of encoding integer variables using RNC

2.1.3 Variable Splitting

Boolean variables and other variables of restricted range can be split into two or more variables [21]. More formally, if a variable x of type T is mapped into n variables of type U by means:

- The splitting (encoding) function $e : T \to U^n$;
- Its inverse $d : U^n \to T$;
- A set of (usually tabulated) operations on U^n so that $e : T \to U^n$ preserves operations.

Example

Boolean values can be split in the following way. Consider a 0/1 representation for truth values, i.e., $x \in \{0, 1\}$, $n = 2$ and $U = \{0, 1\}$. Define $e(x)$ so that

$$e(x) \in \begin{cases} \{(0,0),(1,1)\} \text{ if } x = 0 \\ \{(1,0),(0,1)\} \text{ if } x = 1 \end{cases} .$$

Define

$$d(y_1, y_2) = mod(y_1 + y_2, 2).$$

It can be easily proven that the function $d(y_1, y_2)$ is the inverse of the function $e(\cdot)$. Consider now the matrix $\texttt{AND}(i, j)$:

$$\texttt{AND}(i,j) = \begin{bmatrix} 3 & 3 & 0 & 0 \\ 0 & 2 & 1 & 3 \\ 3 & 2 & 1 & 3 \\ 0 & 3 & 3 & 0 \end{bmatrix}$$

The '&&' operator can be defined in the encoded domain, namely U^2, using the table $\texttt{AND}(i, j)$:

$$\texttt{e}(\texttt{x} \ \&\& \ \texttt{y}) = (\lfloor z/2 \rfloor, mod(z, 2))$$

where $z = \texttt{AND}(2e_1(x) + e_2(x), 2e_1(y) + e_2(y))$. Similar tables can be produced for '||' and '!' operators.
Figure 6 shows the actual `C` programs for the previous example.

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0

int main(int argc, char * argv[]) {

        int x = TRUE;
        int y = TRUE;

        int z = x && y;

        printf("z=%d\n",z);

}
```

$\Rightarrow$

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0

int AND[][4] = {{3, 3, 0, 0},
                {0, 2, 1, 3},
                {3, 2, 1, 3},
                {0, 3, 3, 0}};

int main(int argc, char * argv[]) {

        int x1 = 0;
        int x2 = 1;

        int y1 = 1;
        int y2 = 0;

        int ez = AND[2*x1+x2][2*y1+y2];

        int ez1 = ez / 2;
        int ez2 = ez % 2;

        printf("z=%d\n",(ez1+ez2)%2);

}
```

Figure 6: Boolean splitting example

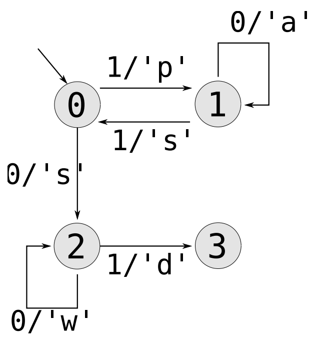
2.1.4 Convert Static to Procedural Data

Convert Static to Procedural Data is an elaborate encoding function proposed for hiding static data such as character strings. The idea is to transform a chunk of static data into some code that once invoked produces the original data [21]. One of the possible implementations is based on Mealy machines as illustrated in Collberg and Nagra's book [20]. A Mealy machine is defined by a set of states $S = \{s_1, ..., s_n\}$, an input alphabet $X = \{x_1, x_2, ..., x_p\}$, an output alphabet $Y = \{y_1, ..., y_q\}$ and two characterizing functions f_y (the output function) and f_s (the transition function):

$$y_k = f_y(x_k, s_k) \tag{2}$$

$$s_{k+1} = f_s(x_k, s_k) \tag{3}$$

A Mealy machine encodes a (partial) function from strings on the input alphabet X to strings on the output alphabet Y and proper invocations to the procedure that implements the Mealy machine can replace usages of static data. An example is reported in Figure 7 along with a possible implementation based on look-up tables.
When the function `gen` is invoked, the formal parameter `wx` is bound to the input sequence. The input is scanned so that each bit is considered a symbol from the alphabet $X = \{0, 1\}$. The current state is held by the variable `state` (initially set to 0). For each state, the two-dimensional matrix



```
...
char out[][2] = {{'s','p'}, {'a','s'},{'w','d'}};
int next[][2] = {{2,1},{1,0},{2,3},{-1,-1}};

char * gen(int wx) {
        int state = 0, in, k = 0;
        char * s = malloc(16);

        while(state != 3) {
                in = wx & 1; wx >>= 1;
                s[k++]=out[state][in];
                state=next[state][in];
        }
        s[k] = '\0';
        return s;
}
...

int main(...) {

  char * p = gen(37);

  // the string pointed by p

  free(p);

}
```

Figure 7: Converting static to procedural data: the Mealy machine and its implementation

`next[][]` tabulates the transition function f_s, so that the element `next[state][in]` defines the next state of the state `state` depending on the value of the input symbol `in`. Correspondingly, for each state, the two-dimensional matrix `out[][]` lists the function f_y so that `out[state][in]` defines which character must be emitted in state `s` when the input `in` is met by the machine. The machine runs until state 3 is reached. A pointer to the produced string is provided as return value. The example presented maps the constant 37 (100101_2) to the string "passwd".

2.2 Aggregation Transformations

Aggregation transformation change how data, both scalar variables and arrays, are arranged in memory.

2.2.1 Merge Scalar Variables

Two or more scalar variables V_1 ... V_n can be merged into one variable W, provided the combined ranges of V_1 ...V_n fit within the precision of W:

$$W = 2^{k_1}V_1 + ... + 2^{k_n}V_n$$

Usually $n = 2$ and we keep this assumption in what follows, i.e.,

$$W = 2^kV_1 + V_2$$

Operations on the original variables can be mapped to operations on the merging variable:

$$\begin{aligned} V_1 = V_1 + d \quad &\Rightarrow \quad W = W + 2^k d \\ V_1 = V_1 * d \quad &\Rightarrow \quad W = W + (d-1)(W \& (2^k - 1)), \end{aligned}$$

where '&' is the bitwise AND operator.

Two int variables (holding positive values) can be fitted into a long:

```
#include<stdio.h>

int main(int argc, char * argv[]) {

  int x,y;
  int d,e;

  x = 31;
  y = 24;
  d = 12;
  e = 5;

  x = x + d;
  y = y * e;

  printf("x=%d, y=%d\n",x,y);

}
```

$\Rightarrow$

```
#include<stdio.h>

#define MASK ((((long)1) << 32)-1)

int main(int argc, char * argv[]) {

  long w; // to store (x,y)
  int d,e;

  w = (((long)31) << 32) + 24; // w = (x,y)

  d = 12;
  e = 5;

  w = w + ((long)d << 32);  // w = (x+e,y)
  w = w + (e-1)*(w & MASK); // w = (x+e,y*d)

  printf("x=%d, y=%d\n",w>>32,w&MASK);

}
```

Figure 8: Merge scalar variables example

2.2.2 Restructure Arrays

Arrays can be reshaped to harden the task of statically determining their content. Confusing the expressions used as indexes and, consequently, obfuscate how elements are accessed is another application of these techniques. Transformations that can be applied to arrays are listed in Table 2.

Transformation	Description
split	an array is split into two or more arrays
merge	two or more arrays are merged into a single one
fold	increase the number of dimensions of an array
flatten	decrease the number of dimensions of an array

Table 2: Restructuring Arrays Transformations

The next paragraphs will provide examples for those transformations.

Splitting

In the example in Figure 9, an array of ten integers is split into two arrays of five integers each, putting items with even index in the first array and items with odd index in the second one.

```
#include <stdio.h>
int x [] = {1,2,3,4,5,6,7,8,9,10 };
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<10; i++) {
        s += x[i];
    }
    printf("sum=%d\n",s);
}
```

⇒

```
#include <stdio.h>
int x1[] = {1,3,5,7,9 };
int x2[] = {2,4,6,8,10};
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<10; i++) {
        s += (i % 2 == 0?x1[i/2]:x2[i/2]);
    }
    printf("sum=%d\n",s);
}
```

Figure 9: Array splitting

Merging

Figure 10 shows array merging, the opposite operation of splitting: Two arrays are merged so that the elements of the first array are put in even-indexed positions while elements from the second array are put in the odd ones.

```
#include <stdio.h>
int x1[] = {1,3,5,7,9 };
int x2[] = {2,4,6,8,10};
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<5; i++) {
        s += x1[i]*x2[i];
    }
    printf("cprod=%d\n",s);
}
```

⇒

```
#include <stdio.h>
int x[] = {1,2,3,4,5,6,7,8,9,10};
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<5; i++) {
        s += x[2*i]*x[2*i+1];
    }
    printf("cprod=%d\n",s);
}
```

Figure 10: Array Merging

Folding

Array folding takes a unidimensional array (Figure 11) and converts it in a two-dimensional array splitting it in two rows. Note that the memory content is not changed, only index expressions to access specific array elements are changed.

```
#include <stdio.h>
int x [] = { 1,2,3,4,5,6,7,8,9,10 };
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<10; i++) {
        s += x[i];
    }
    printf("sum=%d\n",s);
}
```

⇒

```
#include <stdio.h>
int x[][5] = { { 1,2,3,4,5 },
               {6,7,8,9,10}};
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<10; i++) {
        s += x[i/5][i%5];
    }
    printf("sum=%d\n",s);
}
```

Figure 11: Array Folding

Flattening

Array flattening takes a two-dimensional array (Figure 12) and convert it in a unidimensional array by joining its rows. Note that, as in the case of folding, the memory content is not changed.

```
#include <stdio.h>
int x[][5] = { {1,2,3,4,5 },
               {6,7,8,9,10},
               {11,12,13,14,15}};
int main(int argc, char * argv[]) {
    int s = 0;
    int i,j;
    for (i=0; i<3; i++) {
        for (j=0; j<5; j++) {
            s += x[i][j];
        }
    }
    printf("sum=%d\n",s);
}
```

⇒

```
#include <stdio.h>
int x[] = { 1,2,3,4,5,6,7,8,9,10,
            11,12,13,14,15};
int main(int argc, char * argv[]) {
    int s = 0;
    int i,j;
    for (i=0; i<3; i++) {
        for (j=0; j<5; j++) {
            s += x[i*5+j];
        }
    }
    printf("sum=%d\n",s);
}
```

Figure 12: Array Flattening

2.3 Ordering Transformations

Ordering transformations permute items in data structures.

2.3.1 Reorder Arrays

One very naive technique occasionally used by practitioners to try hiding strings embedded in a program from straightforward searches, is to reverse the order of elements in strings. In general any permutation can be applied to array/string elements to achieve a more resilience transformation as shown in Figure 13. Array `p` stores the inverse of the permutation used to shuffle the string "secret message" in the array `m`. As for other types of transformations on strings, a per-character `putchar` must replace the original `printf` function. In general the original string/array must be reconstructed in memory if it has to be passed to any external function, such as `strcmp`.

```
#include <stdio.h>
char * m = "secret message";
int main(int argc, char * argv[]) {
    printf("%s\n",m);
}
```

⇒

```
#include <stdio.h>
char m [] = {
       'r', 'r', 'g', 'e', 'e', ' ', 'm',
       'c', 's', 'e', 's', 's', 'e', 'a' };
char p [] = {
       11, 3, 7, 1, 4, 0, 5, 6, 9, 10, 8,
       13, 2, 12 };
int main(int argc, char * argv[]) {
    int i;
    for (i=0; i<14; i++) {
        putchar(m[p[i]]);
    }
    putchar('\n');
}
```

Figure 13: Array Permutation

3 Selection of Data Obfuscation Algorithms

Section authors:
Roberto Tiella (FBK)

Each technique proposed in previous sections has its own peculiarities, advantages and drawbacks. This section presents a set of features devised to evaluate each technique, an analysis of such features and, finally, the set of transformations selected for implementation.

3.1 Features Grid

To allow an evaluation and a comparison between presented techniques, a set of the most significant features have been collected:

- **Apply to types**: Data type which the technique applies to. Presented approaches are suited for integer types (char, int, long, etc.) and arrays of integer types.
- **Execution time overhead**: An approximate evaluation of loss in speed for the obfuscated program.
- **Memory overhead**: A rough estimation of the memory additionally required by the obfuscated program.
- **Preparation**: Some techniques require the original code to have a specified structure to be applicable. For example, the 'Buffer encoding' technique is applicable to static data only.
- **Homomorphic operations**: Some techniques enjoy the property of being homomorphic on certain operations. Homomorphic operations are specified in such case.
- **Manual post-process**: Some techniques require changes to the source code after they have been applied.
- **Preconditions**: Some techniques impose a set of requirements on the original code. For example, the code must not contain any operation involving pointers to data structure being obfuscated.

Feature values for each technique are reported in Table 3. Values in column 'Preparation' require a further explanation:

- **Points-to information**: Some kind of transformations, for example change encoding, require to know when two variables both point to the same memory location, i.e., the points-to information. In fact, for example, if a pointer p points to the location of an encoded integer variable x, namely $p == \&x$ then:
  - The value returned by the expression $*p$ must be decoded.
  - The value of exp in the RHS of the assignment statement $*p = exp$ must be encoded.

  Static points-to analysis gives a conservative *may-points-to* relation among variables. It specifies when two variable *may* be aliases. There are programs for which precise points-to information can not be computed statically so it would be impossible to know if variables are or are not aliases (reliable pointer analysis is intractable). However, to apply some data obfuscation transformations, precise points-to information is required and the programmer should inspect the output of static pointer analysis and refine it, e.g., by manually adding code annotations with precise points-to and alias information.
- **Identify range**: The interval of values stored in the variable in any execution of the program must be determined. Developers have to provide this information.

Technique	Apply to types	Execution time overhead	Memory overhead	Preparation	Homomorphic operations	Manual post-process	Preconditions
XOR masking	Integral	low	none	Points-to info			
Buffer encoding	Static array	low	none	Prepare static data			
Residue number coding	Integral	high	#variables x 2	Identify range	sum (+), product (∗)		
Variable splitting	Small range integer (typically boolean)	medium	#variables x 2 + 1 matrix per operation (4 range ˆ 2)	Identify range	for operations with matrix		
Convert static to procedural data	Static array	low/medium	buffer length x 3 + malloc size	prepare static string		need to call `free(...)` (to prevent memory leaks)	
Merge scalar variables	Integral	medium	none	Points-to info, Identify sign	sum (+), product (∗) with clear values		
Restructure array (splitting)	Statically allocated array (no malloc)	low/medium	none	array access by index			no pointer arithmetic
Restructure array (merging)	Statically allocated array (no malloc)	low	none	array access by index			no pointer arithmetic
Restructure array (folding)	Statically allocated array (no malloc)	low	none				pointer arithmetic work but prevent obfuscation of index access
Restructure array (Flattening)	Statically allocated array (no malloc)	low	none				pointer arithmetic work but prevent obfuscation of index access
Reorder array	Statically allocated array (no malloc)	negligible	length of vector x index size	prepare static string			no pointer arithmetic

Table 3: Feature values for each specific data obfuscation technique

- **Prepare static data**: Some techniques work on static data only. If in the original code the content of an array is initialized by an algorithm, such code must be replaced with an initialization from precomputed constants.
- **Array access by index**: The program must access array content by index. Bulk operations, such as `memcpy` are not supported.
- **Pointer-arithmetic works but it prevents obfuscation of index access**: Transformations such as array folding and flattening do not change memory content or layout, but their aim is to make it harder to understand which element is indexed. Thus, the use of pointers does not impede the code to be obfuscated, but it makes the obfuscation useless.

Reading the first row in Table 3 it can be seen that XOR masking technique applies to integer types only, namely char, short, int, etc. It has a low execution overhead as the transformation just adds a XOR operation for each access or assignment to a variable. It does not impose any further requirement in terms of memory allocation and, finally, it requires reliable points-to information. The rest of the table describes other transformations presented previously in the section.

3.2 Features Analysis

Analysis of Table 3 reveals the following:

- There are basically two families of techniques based on the 'Apply to types' feature, namely the techniques suited for single-valued variables and those applicable to statically allocated arrays.
- 'Execution time overhead' and 'Memory overhead' features can be aggregated in a single category 'Overhead' with values Low, Memory, Runtime, and Memory+Runtime.
- 'Preparation', 'Manual post-process' and 'Preconditions' can be merged in a 'Manual effort required' classification with values *Yes* and *No*. Reliable points-to information is assumed available in code annotations.

Techniques for single-valued variables

Table 4 shows how the presented techniques for single-valued variables are classified:

- The class of techniques that do not require manual intervention and have a low impact on performance. It contains the XOR masking transformation only.
- A class for techniques that even if they do not imply manual intervention they anyway have an impact on the execution time of the program. It consists of the 'Merge scalar variables' transformation only.
- A class of the more demanding techniques that not only require manual effort at transformation time but that also impose penalties at runtime in terms of memory and execution time. This class collects 'Residue number coding' and 'Variable splitting' techniques.

		Manual effort required	
		No	*Yes*
Overhead	*Low*	XOR masking	
	Memory		
	Runtime	Merge scalar variables	
	Both (M+R)		Residue number coding, Variable splitting

Table 4: Techniques for single-valued variables

Techniques for data arrays

The classification of transformations that apply to data arrays is shown in Table 4:

- 'Flattening' and 'Folding' do not require manual effort and do not penalize the execution.
- 'Buffer encoding', 'Splitting', 'Merging' also do not influence the performance, but they all require some manual intervention on the code.
- 'Convert static to procedural data' and 'Reorder array' that increase memory occupation at runtime and also require manual effort before being applied.

		Manual effort required	
		No	*Yes*
Overhead	*Low*	Folding, Flattening	Buffer encoding, Splitting, Merging
	Memory		Convert static to procedural data, Reorder array
	Runtime		
	Both (M+R)		

Table 5: Techniques for data arrays

3.3 Selection

The implementation of all the state-of-the-art transformations presented in Section 2 goes beyond the scope of the ASPIRE project. The feature analysis of Section 2 represents a good approach to select what are the most promising transformations to implement and integrate in the ASPIRE transformation tool.
The first transformation candidate for implementation is *XOR masking*, because it represents a base case with very low overhead, but with also a quite limited level of protection. Dually, the more advanced *residue number coding* is expected to convey more obscurity to the data, but at the cost of higher memory and runtime overhead. Obfuscation *merge scalar variable* probably stays in the middle of the spectrum, with no memory overhead and just medium runtime overhead. So this transformation is also an interesting case for comparison.
Eventually, a transformation should be also selected among those for data arrays. The one with the largest application scope is probably *convert static to procedural data* as it potentially applies to all the static strings of a program.

Encoding Transformation	Type	NewType
XOR masking	int	int
Merge scalar variables	int	long
RNC	int	long [2] (i.e., an array of type long with length 2)

Table 6: Type mapping in data obfuscation transformations

4 Implementation

Section authors:
Roberto Tiella (FBK)

This section presents a high-level view of how selected data transformations will be implemented as part of the ASPIRE Compiler Tool Chain in the "SLP05: Data Hiding Transformations" component [2]. The state-of-the-art data obfuscation transformation that will be implemented are:

1. XOR masking
2. Variable Merging
3. Residue Number Coding (RNC)
4. Convert Static to Procedural Data

These transformations are implemented as source-to-source transformations. A source-to-source transformation can be viewed as a function defined on the set of C parse trees. In particular a transformation acts on a program parse tree by adding, removing or replacing subtrees. Encoding transformations, specifically, involve global declarations, local declarations and expressions subtrees.

4.1 Rules for Data Obfuscation Transformation

Transformations are written using the TXL programming language. TXL is a rule-based functional language specifically designed for manipulating parse trees. The present section explains how the transformation is performed using a rule-based notation. A rule is defined by two parts: (a) a pattern to match particular subtrees and (b) a replacement for pattern's occurrences found in the code.

4.1.1 Declarations and initializations

Variable declarations must be changed according to the type required to store encoded values.

Rule name	Change type	
Transformation	**Pattern**	**Replacement**
	$Type\ v;$	$NewType\ v';$
Precondition	v is a variable to be encoded	
Description	The type $Type$ of v is changed to $NewType$ according to the storage requirements of the specific transformation applied (XOR masking, RNC, or Merge scalar variables). Storage requirements are described in Table 6. Encoded variables are notated with a prime.	

If the definition statement for a global variable that must be obfuscated presents an initializer, the latter must be encoded.

Rule name	Encode initialization values for global variables
Transformation	**Pattern** $Type\ v = exp;$ **Replacement** $NewType\ v' = ENC(exp);$
Precondition	v is required to be encoded, exp is an expression involving constants only (following C programming language specifications).
Description	$ENC(exp)$ is a constant expression obtained by applying the specific encoding transformation to exp.

4.1.2 Variable Assignments and Uses

Rule name	Assignment to encoded variable
Transformation	**Pattern** $v = exp;$ **Replacement** $v' = ENC(exp);$
Precondition	v is a variable required to be encoded.
Description	v is required to be encoded, thus the representation of v is changed to v' and the expression exp is be encoded.

Rule name	Use of encoded variable
Transformation	**Pattern** v' **Replacement** $DEC(v')$
Precondition	v' is an encoded variable.
Description	An encoded variable is involved in an expression, thus its value must be decoded. $DEC(v')$ is an expression involving v' that evaluates the inverse function of the specific encoding used.

4.1.3 Homomorphic Encoding Functions

If the applied encoding function is homomorphic for a set of operators then a simplification pattern can be applied to expressions involving such operators.

Rule name	Homomorphic operator simplification
Transformation	**Pattern** $ENC(DEC(exp_1) \oplus DEC(exp_2))$ **Replacement** $exp_1 \oplus' exp_2$
Precondition	exp_1 and exp_2 are expressions, $\oplus$ is an operation in the clear domain and $\oplus'$ the corresponding operation in the encoded domain.
Description	The rule is justified by the homomorphic property of the encoding function. $ENC(DEC(exp_1) \oplus DEC(exp_2)) = ENC(DEC(exp_1)) \oplus' ENC(DEC(exp_2)) = exp_1 \oplus' exp_2$

In particular RNC is homomorphic with respect to addition, subtraction and multiplication.

Rule name	RNC Addition Simplification
Transformation	**Pattern** $ENC(DEC(exp_1) + DEC(exp_2))$ **Replacement** $exp_1 +' exp_2$
Precondition	exp_1 and exp_2 are expressions.
Description	Follows directly from the general pattern stated above as RNC is homomorphic w.r.t. addition.

Rule name	RNC Subtraction Simplification	
Transformation	**Pattern** $ENC(DEC(expr_1) - DEC(expr_2))$	**Replacement** $expr_1 -' expr_2$
Precondition	$expr_1$ and $expr_2$ are expressions.	
Description	Follows directly from the general pattern stated above as RNC is homomorphic w.r.t. subtraction.	

Rule name	RNC Multiplication Simplification	
Transformation	**Pattern** $ENC(DEC(expr_1) * DEC(expr_2))$	**Replacement** $expr_1 *' expr_2$
Precondition	$expr_1$ and $expr_2$ are expressions.	
Description	Follows directly from the general pattern stated above as RNC is homomorphic w.r.t. multiplication.	

4.2 Constants

Rule name	Encode constant	
Transformation	**Pattern** $ENC(c)$	**Replacement** c'
Precondition	c is a constant.	
Description	c' is a constant expression that can be evaluated at compile time by the C compiler so that the value of c is not exposed in the binary.	

As an example, consider a snippet of a program consisting in the following statement:

```
x = a + 5*b - c/d;                                    (4)
```

and suppose that the obfuscation requirements state that x, b, c and d have to be encoded using RNC while a is not encoded. The transformation applied on (4) can be decomposed in the following steps:

1. `x = a + 5*b - c/d;`
2. `x' = ENC(a + 5*b - c/d);`
3. `x' = ENC(a + 5*DEC(b') - DEC(c') / DEC(d'));`
4. `x' = ENC(a) +' ENC(5*DEC(b')) -' ENC(DEC(c')/DEC(d'));`
5. `x' = ENC(a) +' ENC(5) *' ENC(DEC(b')) -' ENC(DEC(c')/DEC(d'));`
6. `x' = ENC(a) +' 5' *' b' -' ENC(DEC(c')/DEC(d'));`

because:

- In step 1 the statement matches the “Assignment to encoded variable” pattern and the RHS of the assignment is wrapped by the ENC operator.
- In step 2 the statement matches the “Use of encoded variable” pattern for variables b',c' and d' and DEC operator is applied to those variables.

- In step 3 the statement satisfies the "Addition simplification" and the "Subtraction simplification" patterns so that the expression can be split due to the homomorphic property of RNC w.r.t. addition and subtraction. In Expression 4, $+'$ represent the per-component sum operator in the encoded domain presented in Section 2.1.2.
- In step 4 the statement matches the "Multiplication simplification" pattern and for the same reason of the previous step the expression involving the multiplication is split.
- In step 5 the statement satisfies the "Constant encoding" and the "Identity" patterns. In expression 6, $5'$ stands for the constant expression that evaluates to the encoding value of 5 in RNC.

Finally, once that specific parameters for RNC are chosen (see Section 2.1.2), e.g., $u = 2$, $m_1 = 31$ and $m_2 = 33$, statement 6 is rewritten as the following two C statements (declaration of variables are included for clarity):

```
long b[2], c[2], x[2];
int a;

x[0] = (a % 31) + 36 * b[0] - (dec(c)/dec(d)) % 31;
x[1] = (a % 33) + 38 * b[1] - (dec(c)/dec(d)) % 33;
```

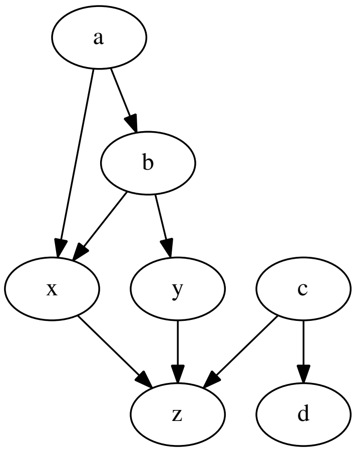
where `dec` is a function that implements the decoding function of RNC.

5 Balancing Security and Performance

Section authors:
Mariano Ceccato, Roberto Tiella (FBK)

As underlined in Sections 2 and 3 data obfuscation transformations cannot be applied for free. Gain in obscurity and loss in runtime performance must be traded and an obfuscation strategy must be devised. The most conservative strategy to prevent code tampering is *obfuscate all*, i.e., to

```
1     //safe vars
2     int a,b,c,d;
3
4     //sensitive vars
5     int x,y,z;
6
7     a = 2;
8     b = a + 3;
9     x = a + b;
10    c = 12;
11    y = b + 1;
12    z = c + y;
13    d = 2 * c;
14    z = x;
```



Note: x, y and z are sensitive variables while a, b, c and d are not.

Figure 14: An example of C snippet and the related DPG

apply obfuscation to all the program variables. However, not all the variables in a program are expected to be security critical, e.g., variables related to the GUI might not represent a security threat in case of tampering. Thus, such an aggressive approach could cause unmotivated and unacceptable performance degradation. The opposite strategy is *obfuscate just sensitive*. It consists of obfuscating only the particular program variable(s) that is (are) security sensitive and is (are) prone to attacks, e.g., those variables in strategy games that store gold and energy values. However, other variables that are not intrinsically sensitive could be somehow related to sensitive variables. Related variables could leak important information that could be potentially used by the attacker to guess or tamper with the value of a sensitive variable.
With respect to the example of Figure 14, we see that at line 11 variable `b` is assigned to variable `y`. If an attacker knows or tampers with the (clear) value of `b` before line 11 , the value of `y` is know or tampered with when the execution reaches line 11, even if `y` is obfuscated. Thus, the strategy *obfuscate just sensitive* is also suboptimal, in fact related variables should also be considered for obfuscation. Intuitively, values that participate to the definition of a sensitive values should be also obfuscated, so they should be variables that are defined using sensitive values. To capture this intuition, we propose to consider neighbourhoods of certain size around sensitive variables in the *Data Proximity Graph* as presented in the next section.

5.1 Data Proximity Graph and Data Distance

Given two variables v_1 and v_2, we say v_1 is *proximal* to v_2, written as $v_1 \to v_2$, if v_1 is used to define the value of v_2. In the example of Figure 14, because of the assignment on statement 11 `y=b+1`, variable `b` is proximal to variable `y`, i.e., $b \to y$.
Given a program, the Data Proximity Graph (DPG) for the program is a directed graph where nodes are the variables $\{v_i\}$ and there exists an edge between nodes v_1 and v_2 iff $v_1 \to v_2$, namely

if v_1 is proximal to v_2. The complete DPG for the example is reported in Figure 14.
Informally, on the DPG, the distance between two nodes is the number of assignments that we have to traverse to use the first variable in an assignment of the second one. Using the DPG it is possible to define a distance, called *data distance* between variables in a program. More formally the distance between two variables v_1 and v_2 is defined as

$$d(v_1, v_2) = min\{|P| : P = path_{DPG}(v_1, v_2)\} \tag{5}$$

In our running example, there is an edge from `z` to `c`. For this reason data distance $D(z, c)$ is 1. From this graph we can see that the distance between `z` and `b` is 2, the distance between `b` and `a` is 1 and the distance between `z` and `a` is 2. In Figure 15 (on the left) a table is shown listing distances between variables for program in Example 14.

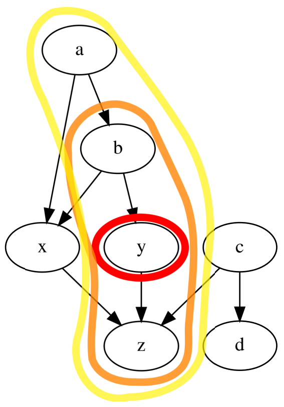
5.2 Neighborhood of a variable

Given a (sensitive) variable v, the *neighbourhood of v with radius r*, $N_r(v)$, can be defined as:

$$N_r(v) = \{v'|d(v, v') \leq r \vee d(v', v) \leq r\}.$$

In other words, $N_r(v)$ is the set of variables that reaches v or are reached by v in r assignments. Figure 15 shows neighbourhoods of y with radius 0 (in red), 1 (in orange) and 2 (in yellow). Remaining variables cannot reach y or cannot be reached by y. Given a sensitive variable v, its neighbourhood will contain more and more variables as the radius increases, defining a growing sequence of subsets of variables that could be obfuscated to increase the level of data obscurity of the code v belongs to.

	c	z	y	d	x	a	b
c	0	1	-	1	-	-	-
z	-	0	-	-	-	-	-
y	-	1	0	-	-	-	-
d	-	-	-	0	-	-	-
x	-	1	-	-	0	-	-
a	-	3	2	-	1	0	1
b	-	2	1	-	1	-	0



On the left: the matrix of distances between variables for program in Example 14. In color distances used to define the neighbourhoods on the graph depicted on the right side of the figure. On the right: neighbourhoods of y with radius 0 (in red), 1 (in orange) and 2 (in yellow).

Figure 15: Variables data distance matrix and neighbourhoods of a sensitive variable

5.3 DPG Implementation

A component to compute the Data Proximity Graph was implemented as an extension of CodeSurfer [1]. CodeSurfer is intended to support developers in program-understanding of C/C++ code. Starting from source code, it generates a variety of graphs and views, that can be interactively browsed and customized. CodeSurfer performs source code analysis and, among other analyses, this tool generates the System Dependence Graph (SDG). Analysis results are exposed to a Scheme API that can be used to extend the tool or to implement custom analyses as batch scripts. The computed Data Proximity Graph is then passed to a graph visiting algorithm that extracts the neighbourhood of a given radius for a given variable.

6 Experimental Assessment

Section authors:
Roberto Tiella (FBK)

Given a sensitive variable v, the radius r of its neighbourhood can be used as parameter to numerically describe the obfuscation strategy and consequently the strength and expected overhead for data obfuscation. This section presents an empirical investigation of the impact on the performance of a program of changing the radius of the neighbourhood of a given variable. In particular, we describe results of an experiment devoted to compare the performances of the three transformations devised to obfuscate integer variables, namely *XOR masking*, *Merge scalar variables* and *RNC*.

6.1 Metrics

The assessment is based on a set of metrics that were collected by two main means: (a) dynamically measuring performances at runtime and (b) statically inspecting the source code or the compiled binary of the subject program. Each metric is described in the rest of this subsection.

IENC/IDEC:

The number of encoding/decoding function invocations (IENC/IDEC) is computed by instrumenting the obfuscated code. Every time an encoding or decoding operator is applied a related counter is incremented and counter values are printed before exiting the main function. Different input can cause different execution path to be followed by the program and consequently a different number of invocations to encoding/decoding function. Therefore measures are averaged across a fixed number of executions.

ETIM:

The execution time (ETIM) is measured by means of the `time` utility. In Unix systems, the time used by a process to execute is accounted as *system time* and *user time*. We define ETIM as the sum of system and user time.

SMEM:

The size of memory allocated to variables for the program (SMEM). It was computed in the following way: (a) all local variables are moved to the global scope; (b) the code is compiled; (c) the executable is inspected by means of the tool `objdump -s`; and (d) bytes are counted for entries of type `.data` (initialized static memory) and `.bss` (zero initialized static memory).

NOBV:

The obscurity of the code is represented by the amount of the program that is subject to data obfuscation. After identifying a sensitive variable that must be obfuscated, we gradually increase the *radius of the neighbourhood* to include more and more variables in the set of variables to obfuscate.

6.2 Case study

The assessment was performed on the *License check* subject. License check is a routine devoted to check the validity of a license number to activate a software component. The serial number contains the date when the license has been emitted; its validity is meant to expire 30 days after emission. The security sensitive variable is the one that holds the difference in days between emission date and current date. In fact, an attacker might tamper with this value to make an

expired license last longer. An attacker might (i) add a constant value to the current date, (ii) subtract a constant value from the difference, or (iii) add a constant value to the date extracted from the license number. All these examples are instances of attacks based on data tampering that could be mitigated using data obfuscation.
Of course many other attacks are possible based on other strategies, such as tampering with the code to skip license validation, altering the system date, or tamper with the system library that fetches the current date. All these attacks are out of the scope of data obfuscation. Different protections are effective against these attacks, such as code obfuscation or remote attestation. The difference between current date and date from license should be obfuscated. However, other variables involved in the computation might leak sensitive information, such as dates, days, months, years, and they are also candidate for obfuscation according to their distance to the sensitive variable.

6.3 Experimental Setting

To study the impact of obfuscation, a set of execution scenarios have been defined for the case study. We defined 100 scenarios, corresponding to licenses emitted on different dates. Half (50/100) of the licenses are valid, the reset (50/100) of them are expired.
As the execution of the original license check program was quite fast, we artificially modified the program by introducing an iteration of $50 \cdot 10^6$ executions of the *main* function. This way, we are able to have the program execution lasts for tens of seconds, improving the precision for the measurement of ETIM.
The experimental process consists of running the original (clear) code on the execution scenarios, to collect ETIM. Then, for increasing number of variables to be obfuscated and for each obfuscation technique, we apply data obfuscation, we measure SMEM on the executable, and we execute the obfuscated code to collect ETIM and INVE/D values. To make sure that data obfuscation preserves the original semantics, on all the scenarios output of the obfuscated code is compared to the output of clear code.
The experiment has been conducted on a Desktop with Intel Xeon 3.3 GHz CPU (4 cores), 16 GB of memory, running Red Hat 6.5 64 bit.

6.4 Results

6.4.1 Number of obfuscated variables

Table 7 lists the number of obfuscated variables (NOBV) for different values of the radius r of the neighbourhood around the variable chosen as sensitive. NOBV varies from 1 to 10. In particular for any radius r greater than 3, NOBV is 10.

Radius (r)	NOBV
0	1
1	3
2	8
≥ 3	10

Table 7: Number of obfuscated variables in a variable neighbourhood

6.4.2 Invocations

Table 8 reports the average number of invocations to encoding and decoding functions given the number of obfuscated variables and obfuscating technique used. XOR masking and Merge scalar variables requires the same number of invocations. Invocations for RNC are quite different. In particular, the average number of invocations of the decoding operator for the RNC is much smaller than the corresponding number for the other two techniques (9.4 vs. 21). This is an empirical evidence that RNC imposes a limited number of decoding operations if homomorphic operators, i.e., addition, subtraction and multiplication, are involved.

	NOBV							
	1		3		8		10	
Technique	IENC	IDEC	IENC	IDEC	IENC	IDEC	IENC	IDEC
XOR masking	1	1	5	6	22.4	17	26.4	21
Merge scalar variables	1	1	5	6	22.4	17	26.4	21
RNC	4	1	16	1	20	5.4	26	9.4

Table 8: Number of encoding/decoding operation invocations

6.4.3 Memory overhead

Table 9 shows the numbers of bytes statically allocated to variables for a range of obfuscated variables. As expected, XOR masking does not present any memory overhead (57 bytes is the memory required also by the original program). Variables merging technique requires 4 bytes more when the number of obfuscated variables is odd. *RNC* consumes 12 bytes more for each new variable that requires to be obfuscated (300 % of memory consumption).

	NOBV			
Technique	1	3	8	10
XOR masking	57	57	57	57
Merge scalar variables	61	61	57	57
RNC	69	93	153	177

Table 9: Memory (bytes) allocated depending on the number of obfuscated variables

6.4.4 Runtime overhead

Table 10 shows average execution time (and standard deviation in parentheses) for the set of techniques used to obfuscate and the number of obfuscated variables. Execution times are also plotted in Figure 16 to make trends more clear. In the plot, vertical bars show the variability of the measure, dots correspond medians, lower small horizontal bars correspond to 0.25 quantiles and, finally, upper small horizontal bars correspond to 0.75 quantiles.
It can be seen that the impact of XOR masking is negligible. Applying Merge scalar variables technique resolves in slowing the execution down by just a small percentage (2.4 % as a maximum). The impact of RNC is quite evident reaching about 20% when 10 variables are obfuscated.

	NOBV				
Technique	0	1	3	8	10
none	42.14 (0.15)				
XOR masking		42.19 (0.27)	42.24 (0.15)	42.53 (0.17)	42.57 (0.15)
Merge scalar variables		42.23 (0.15)	42.47 (0.16)	43.13 (0.15)	43.24 (0.19)
RNC		42.99 (0.22)	44.72 (0.17)	48.15 (0.27)	51.52 (0.25)

Table 10: Average execution times by technique and number of obfuscated variables.

Table 11 reports Pearson correlation and linear regression coefficients between NOBV and ETIM per technique used. All cases are statistically significant with a p-value < 0.01. As observed on the graph, ETIM reports a significant correlation with NOBV, $\rho > 0.7$ for XOR masking and close to 1 for Merge scalar variables and RNC.

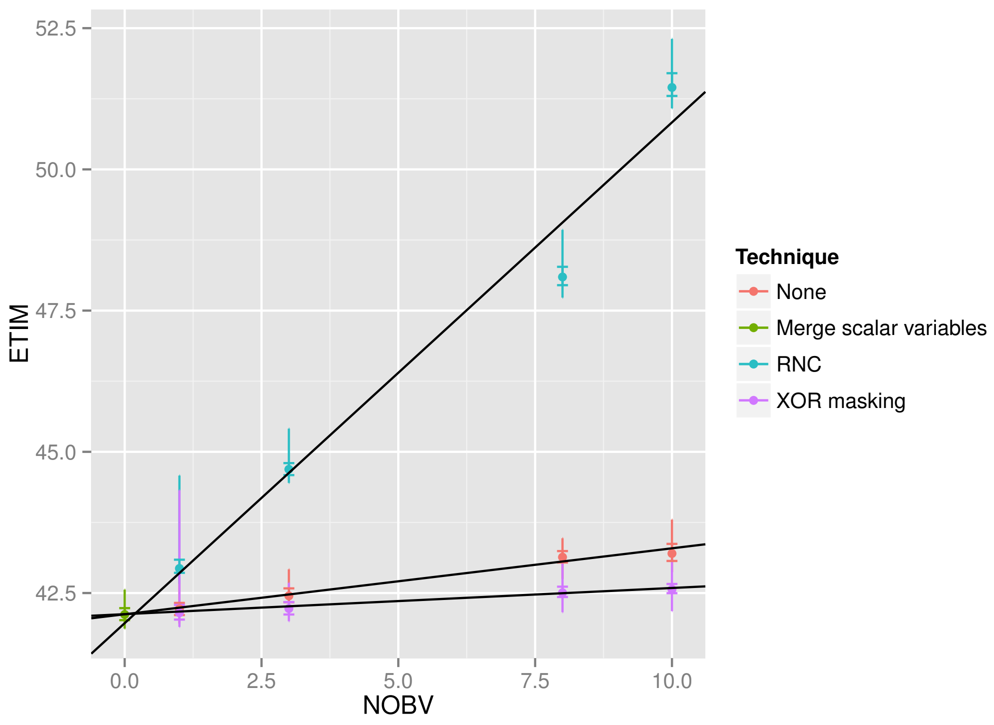


Figure 16: Execution time per technique varying the number of obfuscated variables

Technique	P-value	Correlation (ρ)	LM Coefficient
XOR masking	<0.01	0.70	0.05
Merge scalar variables	<0.01	0.94	0.12
RNC	<0.01	0.99	0.88

Table 11: Pearson correlation and linear regression coefficients between NOBV and ETIM.

6.5 Conclusions

As expected, the experimental validation confirms that different obfuscation techniques have different impacts on performance both in terms of used memory and execution time. In particular, as listed in Table 12 XOR masking has no impact on used memory and a negligible impact on execution time. Merge scalar variables doesn't impact on memory usage and it has an almost negligible impact on execution time (about 2.5%). RNC is the most demanding transformation with an increase of 300% of memory consumption (for obfuscated variables) and an execution time degradation of 20%.

Technique	Memory Overhead	Execution Time Overhead
XOR masking	0	negl.
Merge scalar variables	almost 0	$<$ 2.5%
RNC	300 %	20 %

Table 12: Empirical assessment results for obfuscation overhead.

7 Related Work

Section authors:
Mariano Ceccato, Roberto Tiella (FBK)

Data obfuscation transformations are described and qualitatively analyzed in various previous works [22], [21]. Drape at al. describe how data obfuscation transformations can be formalized using data refinement, a setting that allows to prove transformations correctness formally [32]. Residue Number Coding and its homomorphic properties are presented by Zhu, W. and Thomborson C. [73]. Their study concerns theoretical properties of the RNC transformation. The same authors published an article on applying homomorphic data obfuscation to array indexing [74]. All these works present definitions, applications and qualitative analyses for data obfuscation techniques in general, and homomorphic data obfuscation in particular. However, these works do not address the actual implementation of the proposed obfuscation schemes. Consequently, no empirical assessment has been conducted to evaluate applicability of data obfuscation in real world programs.

In the past, the assessment of obfuscation transformations has been conducted by measuring the complexity introduced by obfuscation mainly through code metrics. However, their objective was mainly to measure the effect of *code* obfuscation on program statements, rather than the effect of *data* obfuscation on data structures.

8 Status and Plan for Task 2.1

Section authors:
Mariano Ceccato (FBK)

During the first year of the project we focused on the implementation of four state-of-the-art schemes to data obfuscation. These schemes are xor integer encoding, residue number coding, merge scalar variable and convert static to procedural data. At the moment these four schemes are implemented and fully working on a case study (i.e., license check) that has been used to quantify overhead caused by obfuscation.
In the second year of the project, we plan to apply data obfuscation to the project case studies and assess the obfuscation on real world code. Moreover, we will focus on the elaboration of novel extensions of these transformations. In particular, the current transformations are vulnerable to static attacks. In fact, obfuscation parameters (e.g., the mask for xor integer encoding and the bases for residue number coding) are present in the code and they could be spotted by an attacker. Once these parameters are known, the obfuscation can be easily subverted.
To overcome this limitation, in the second year, we will investigate extensions that would turn static parameter values into dynamic values, that would be more difficult to guess. For example, obfuscation parameters could become opaque constants, i.e., values that depend on complex alias conditions on a pointer intensive data structure. In this way, obfuscation parameters would be difficult to guess using static analysis, because precise points-to analysis is known to be intractable.
The final version of the data obfuscation support will be delivered in D2.07 and described in D2.08 at month M24.

Part II
White-Box Cryptography

9 Introduction to White-Box Cryptography

Section authors:
Brecht Wyseur, Patrick Hachemane (NAGRA)

The traditional goal of cryptography is to protect communication. In the past 30 years, many cryptographic protocols and ciphers have been designed to protect communication against eavesdropping and to ensure it integrity and authenticity. Examples are the TLS protocol [27], AES and (T)DES block ciphers [24, 25], or RSA and ECC asysmmetric ciphers [58]. In their design considerations, the end points are assumed trusted – the attacker only has access to the input/output of the algorithm (black-box). For such a model to comply, the algorithm needs to be exected in a secure environment.
In the ASPIRE project, we consider the MATE attack context, where cryptographic algorithms are deployed in applications that are executed on open devices, as described in Deliverable D1.02 ("the ASPIRE attack model"). In literature, this is also sometimes refered to as the *white-box attack context* [19]. In such a context, the implementation itself is the sole line of defense. The challenge that white-box cryptography aims to address is to implement cryptographic ciphers in such a way that they do not leak critical information such as cryptographic keys. We denote such implementations as *white-box implementations*.

9.1 State of the art

White-box cryptography has been introduced in 2002 with the seminal papers of Chow *et al.*, which present a white-box DES implementation and a white-box AES implementation respectively [19, 18]. They present a technique to hard-code the cryptographic key into the implementation of the block cipher, by transforming the entire block cipher representation into a sequence of key-dependent lookup tables. The secret key is hard-coded into these lookup tables and protected by some special-purpose randomization techniques.
The white-box DES implementation was the first that was shown to be insecure. Its vulnerability is mainly due to the DES Feistel structure which can be distinguished in a lookup table representation. The first white-box cryptanalysis techniques find their origin in side-channel attacks such as fault propagation correlation [53] or guess & determine attacks [56]. However, these techniques make some assumptions on the implementation that can easily be mitigated [56]. It was only in 2007 that the whitebox DES implementation was completely broken with the truncated differential cryptanalysis by Wyseur *et al.* [69]and Goubin *et al.* [49].
The white-box AES implementation has been broken by Billet *et al.* [9] using algebraic cryptanalysis techniques, which was further improved to the general case of SPN ciphers by Michiels *et al.* [59]. Wyseur showed how these algebraic attacks can be used to attack any implementation that comprises lookup tables [68].
The algebraic cryptanalysis techniques led to the design of new constructions beyond the lookup table strategy, with directions towards implementations with randomized algebraic equations [7] and the introduction of perturbations to defeat the algebraic structure that enables to mount algebraic attacks [14]. The implementation by Billet and Gilbert [7] was broken due to a cryptanalysis result of the underlying primitive that was exploited. Attempts to resolve this issue were inspired by Ding to include perturbation functions to *destroy* the algebraic structure [28]. This led to an improved traceable block cipher scheme [13] and was eventually applied to white-box AES implementations [14]. Nevertheless, even these new improved constructions have shown to be insecure by De Mulder *et al.* [62]. The last advancements in the area include a new proposal by

Xiao and Lai [70] which was subsequently broken by De Mulder *et al.* [60], and a proposal by Karroumi [55] based on dual ciphers that was broken by De Mulder *et al.* [61].

9.2 WBC activities in the ASPIRE project

In the ASPIRE project, we aim to advance the state of the art in WBC in different directions.
First, since WBC requires dedicated transformations to hard-code cryptographic keys into implementations or to enable white-box implementations that can be instantiated *dynamically*, tools are required. We aim to design and implement tools that generate and manage white-box code and that can be interfaced with the ASPIRE tool flow. The progress on this activity is reported in Section 10.
Secondly, we investigate new directions in white-box cryptography to mitigate the attacks as described in the state of the art. We aim to find new techniques based on multivariate cryptography and fully homomorphic cryptography. The results of this research activity are described in Section 11.
The third activity of Task 2.2 covers the research and development of white-box implementations that can be of practical use for the use-cases envisioned in the ASPIRE project. This includes the design of dynamic white-box implementations, and time-limited white-box implementation. The latter refers to white-box implementations that have a trade-off between performance and security that rather benefits peformance to meet those requirements that have been described in Deliverable D1.03 ("Security Requirements"). The results of this activity are described in Section 12.

10 White-Box Tool for ASPIRE (WBTA)

Section authors:
Brecht Wyseur, Patrick Hachemane (NAGRA)

10.1 Introduction

As explained in Section 9, white-box cryptography (WBC) cannot be considered as a definitive solution against piracy. To increase security robustness and duration, WBC must be segmented across markets, operators or even users (space diversity); moreover, it must be renewed, ideally in anticipation of attacks (preventive action) or as a swift response to attacks (corrective action). In order to meet these objectives, NAGRA developed a framework to replace standard (vanilla) cryptography by WBC in a transparent and automated way. This tool, denoted as the White-Box Tool (WBT) has been delivered to the consortium and comes with a reference guide that has been put available to the consortum as ASPIRE D2.04 [3]. The WBT comes with many features, including performance benchmarking of generated code, testing, management of crypto generations, insertion of tag values, generation of server support files, compilation of client object files, templates, etc. As a result, the WBT comes with a complex interface. To ease the integration in the ASPIRE Compiler Tool Chain (ACTC), NAGRA has developed a frontend to the WBT that presents a simplified interfaces for the ACTC. This component is denoted as the WBT for ASPIRE (WBTA). The WBTA has also been delivered to the consortium and a reference guid has been put available as ASPIRE D2.04b [4].
Under the hood, the WBTA uses the input data that it received from the ACTC to prepare the appropriate input for the WBT. The WBTA functions as a front-end (or wrapper) to the WBT: it will invoke the WBT with the appriorate input and manage its return for the ACTC. Nevertheless, the WBT remains functional without the WBTA and can be called directly.
In the following sections, we provide more details on the WBTA elements and present the current status.

10.2 Overview

The WBTA is a tool written in Python, that can be invoked by the ACTC via the API as described in the internal reference guide D2.04b [4]. The WBTA needs to be invoked by the ACTC each time some cryptographic function needs to be replaced with a white-box cryptographic primitive. This step is captured as step SLP03.02 in ASPIRE Deliverable D5.01 [2]. Two different inputs are needed. The first one, called source input, is related to the cryptographic function to replace (what to replace); it is retrieved from source annotations by the src2src tool. The second input, called decision input, is related to the way this function must be replaced (how to replace). The ADSS is in charge of selecting the protection to apply, given the available white-box crypto primitives, the desired level of security, the acceptable code size, and the crypto operation speed; the exact way these parameters are transmitted from ADSS to WBTA is not yet defined and is out of scope of the present report. Figure 17 depicts the process flow as currently implemented with the ACTC and the WBTA, where the decision input is hard-coded in a file used by the WBTA. Figure 18 depicts the process that is envision in subsequent releases, where the ADSS will be integrated to provide configuration options. In both figures, what is depicted as orange boxes are the white-box components that are developed within task 2.2.

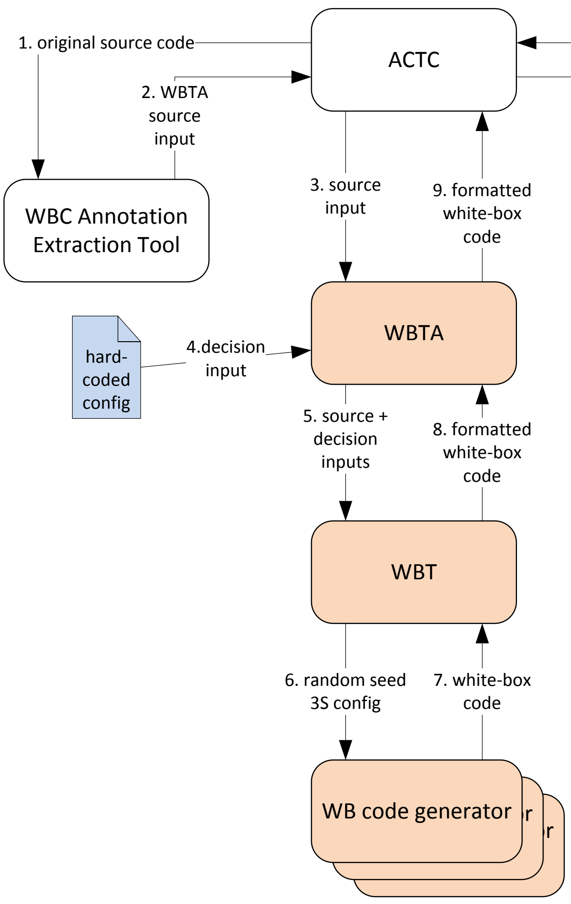


Figure 17: WBTA in Aspire context, initial phase

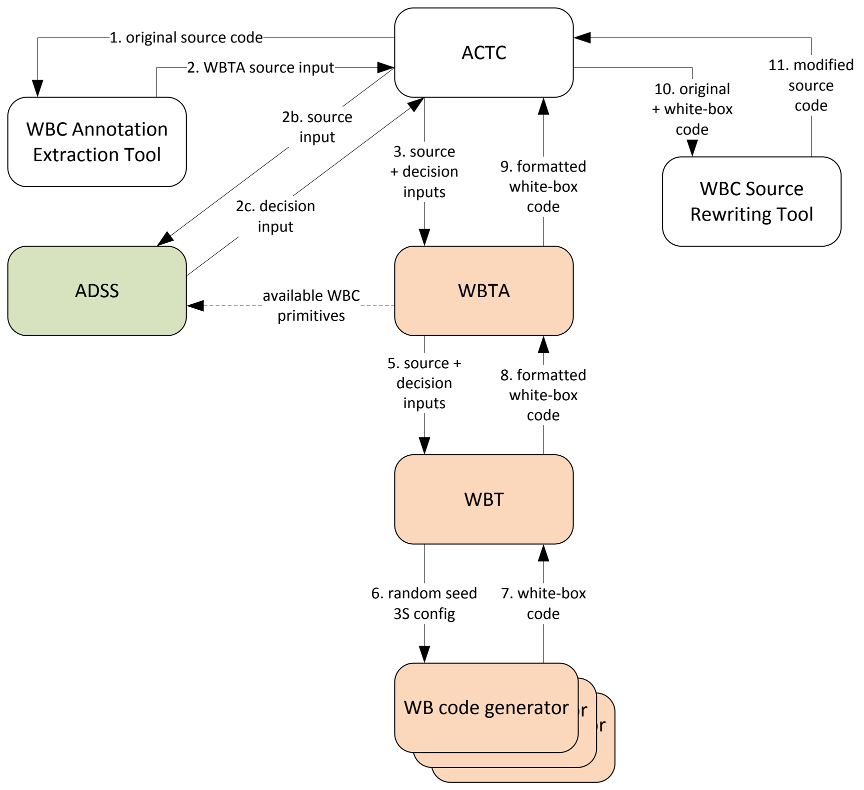


Figure 18: WBTA in Aspire context, subsequent phases (draft)

The configuration input to the WBTA is provided in an XML format as specified in D2.04b [4], the result is a C source code fragment that is to be inserted into source files; this code needs to replace the original cryptographic code. The WBT and WBTA produce additional operational reporting information that can be written to stdout or to a log file.

10.3 Annotations

FBK and Nagra agreed on the format of the annotations used to identify a cryptographic function to be replaced by a white-box crypto primitive. The format of the annotations are presented in ASPIRE Deliverable D5.01 [2]. In addition to mark the call to the cryptographic function, the annotations also have to mark the fixed key used during this call, since this key must be removed from the source code. We refer to the next section for additional information.

10.4 Source input

The source input is retrieved by the src2src tool, from the annotations in the source file as described in ASPIRE Deliverable D5.01 [2]. The source input is composed of:

- *key size:* the size of the key, in bytes;
- *key value:* the value of the key; applies only to a fixed-key implementation;
- *iv:* the value of the initialization vector; applies only to CBC modes;
- *algorithm:* the name of used crypto algorithm (aes, des, tdes, rsapriv, rsapub);
- *operation:* the operation performed by the algorithm (encrypt or decrypt);

- *mode:* the mode of the used algorithm (ECB, CBC, CBC_INV);
- *label:* a label to uniquely identify this case; this label is used to build the name of the function.

10.5 Decision input

The decision input is determined by the ADSS, according to the constraints on the application size, speed and security. It is composed of:

- *random seed:* a random seed of 32 bytes;
- *primitive:* the WBC primitive to use in replacement of the original cryptographic function;
- *size:* the configuration of the WBC primitive regarding code size;
- *speed:* the configuration of the WBC primitive regarding code speed;
- *security:* the configuration of the WBC primitive regarding code security.

As discussed earlier and presented in Figure 17, the decision input is at this phase in the project hard-coded into a configuration file.

10.6 Output

The output produced by WBTA is composed of:

- *client code:* the code of the WBC primitive replacing the original cryptographic function;
- *server code:* for a dynamic-key implementation, the code used on the server side to protect a key.

Ideally, the code should be produced as a source code fragment intended to replace the original function. In practice however, the complexity white-box code forces the delivery to be composed as dedicated functions that may be implemented in one or more source code files that need to be linked with the application. This is a more pragmenatic approach than code fragments, because white-box code may be quite large, use sub-routines, define additional constants that are defined in dedicated header files, include macro's, inline functions, etc. This set of produced source files is defined as "SC04.01" in Deliverable D5.01.
In this phase of the project, we will assume that the produced code is composed of 1 source file and comprises one public functions that needs to be invoked. This function call needs to replace the call to the original cryptographic function. In subsequent releases of the WBTA and ACTC, we will specify how an arbitrary number of source files can be included, and how to deal with code that is shared by multiple white-box implementations and is thus redundant.

10.7 Status

As of today, the WBT is fully implemented and validated on the build platform that is selected for the ASPIRE project (we often refer to *the ASPIRE VM* as this build platform, as it concerns a Virtual Machine image that is shared between all consortium members). The reference guide of the WBT and the WBTA are both available in internal ASPIRE documents [3, 4]. The delivery of the WBTA follows with the integration into the ACTC as described in the ASPIRE Deliverable 5.03 – the report on the first ACTC implementation. Support for the integration into the ACTC is ongoing, as well as improvements to facilitate smooth integration. The full release of the WBTA is planned by M18, in accordance to the schedule as described in the ASPIRE Deliverable of Work.

11 New WBC Schemes with Provable Security

Section authors:
Brecht Wyseur, Patrick Hachemane (NAGRA)

As previously introduced, the initial white-box implementations presented by Chow *et al.* [18, 19] present a technique to hard-code a secret key into lookup tables and then protect these by applying some special-purpose randomization technique. In general, the approach in WBC is to blend together the fixed key and random data in such a way that they cannot be unblended any more. This random data is generated at code generation time and introduced in such a way that it preserves the overall semantic behaviour of the implementation.
The basic idea works as follows. Consider a program P that comprises a sequence of 3 lookup tables L_0, L_1, and L_2 that are executed subsequently ($P = L_2 \circ L_1 \circ L_0$), where L_1 comprises some secret key information that needs to be hidden (e.g., $L_1(x) = f(x)^k$ for some arbitrary function f). One may *obfuscate* this program by generating random bijective functions F_0, F_1 that transform P by randomizing each lookup table as follows:

$$\begin{cases} L_0 & \rightarrow & L'_0 = F_0 \circ L_0 \\ L_1 & \rightarrow & L'_1 = F_1 \circ L_1 \circ F_0^{-1} \\ L_2 & \rightarrow & L'_2 = L_2 \circ F_1^{-1} \,. \end{cases}$$

The result is program $P' = L'_2 \circ L'_1 \circ L'_0$ that is semantically equivalent to P, but becomes harder to analyse. Looking at L'_1: in order for an attacker to gain information on this lookup table that may lead to obtaining the secret information k, he will first need to obtain knowledge of F_0 and F_1. In the case where L_1 is a bijective mapping, and given only information on L'_1, this will be impossible – L'_1 enjoys the property of *local security*. This is similar to the security of the one-time pad. To extract information the attacker will need to extract information from L'_0 and L'_2. The goal of this approach is to push the adversary to an analysis that is equivalent to a black-box attack (analysis of the semantic behaviour of P').
The whole idea pursued relates to the decomposition problem of lookup tables. While this problem holds for single bijective lookup tables, it does not hold for non-bijective lookup tables (as shown by Wyseur [68]) nor for groups of lookup tables [9, 59, 68]. Therefore, it is very challenging to define more robust constructions based purely on lookup tables and its security cannot be proven. The approach as conceived by Chow *et al.* remains ad hoc. Nevertheless, it does not exclude that similar concepts on other mathematical primitives can be defined, or other related concepts can be relied upon. This is the objective of the research on new cryptographic WBC schemes that are provably secure. In particular, we will build on recognised hardness problems (such as the functional multivariate polynomial decomposition problem) that mitigate algebraic attacks as presented before and upon which we can define concrete security properties. In this section, we report about our progress in this task.

11.1 Multivariate WBC

11.1.1 Functional Multivariate Polynomial Decomposition Problem

The new direction that we conceive in this section concerns white-box implementations whose security relies on the functional multivariate polynomial decomposition problem, which is expressed in Definition 1 [26].

Defininition 1 (Functional Multivariate Polynomial Decomposition Problem (FDP)) *If K is a field, and $f, g \in K[x_1, \ldots, x_n]$ are multivariate polynomials, then $h = f \circ g = f(g) \in K[x_1, \ldots, x_n]$ is their (functional) composition. (f, g) is a (functional) decomposition of h.*
The functional multivariate polynomial decomposition problem *over the extension field $K[x_1, \ldots, x_n]$ can be stated as follows: given $h \in K[x_1, \ldots, x_n]$, determine whether there exists a decomposition (f, g) of h with f and g of degree greater than one, and in the affirmative case, compute one.*

Note that when such a decomposition exists, it cannot be unique. Indeed, consider $h = f \circ g$, then any bijective linear combination A leads to a decomposition of h since $h = (f \circ A^{-1}) \circ (A \circ g)$. It has been shown by Dickerson in 1989 that the multivariate polynomial decomposition problem is NP-hard [26].

Defininition 2 (Multivariate Map) *A system of multivariate polynomials f is a multivariate map mapping from $K[x_1, \dots, x_n]$ to $K[y_1, \dots, y_m]$, which comprises m multivariate polynomials $f_i \in K[x_1, \dots, x_n]$:*

$$f = \begin{cases} f_1(x_1, \dots, x_n) & = \sum_{e_1, \dots, e_n} \alpha_i \cdot x_1^{e_1} \cdots x_n^{e_n} \\ & \vdots \\ f_m(x_1, \dots, x_n) & = \sum_{e_1, \dots, e_n} \alpha_i \cdot x_1^{e_1} \cdots x_n^{e_n} \end{cases}$$

The degree of the multivariate mapping is the degree of the highest order term $\max(e_1 + e_2 + \cdots + e_n)$*; the highest degree of any f_i.*

The composition of multivariate polynomials $h = f \circ g$ can be extended to the composition of systems of multivariate polynomials $h = (h_1, \dots, h_u) == (f_1(g_1, \dots, g_n), \dots, f_u(g_1, \dots, g_n))$, by replacing each ith variable in S_f by the ith multivariate polynomial in S_g. Hence, the number of polynomials in S_g must be equivalent to the number of variables over which the multivariate polynomials in S_f are defined.
The functional multivariate polynomial decomposition problem can be translated easily onto the functional decomposition problem of systems of multivariate polynomials.

11.1.2 Our contribution

The main objective in our approach is to define cryptographic algorithms as a series of multivariate polynomials S_i (e.g., where each system embodies one round of the cipher), and then to obfuscate the cipher by including random systems of equations R_i, in such a way that from the resulting system of equations $O_i = R_{i+1} \circ S_i \circ R_i^{-1}$, no information on S_i can be obtained.

Challenges

There are a number of challenges in this approach that need to be addressed.

- First, the cipher needs to be representable in an algebraic form to be able to define such systems of multivariate equations that represent the same functionality.
- When the system S_i gets 'obfuscated', the size of the system of polynomials may explode and its performance be degraded beyond practical purposes; even if the cipher originally could be expressed as small equations (before obfuscation). The latter is for example the case with the AES.
- The security of the result must be evaluated. In the past few years, there has been a vast amount of research on multivariate polynomial decomposition and algebraic cryptanalysis techniques. We need to investigate what configurations may lead to secure implementations.

Hence, for our research to lead to practical results, we need to use 'white-box friendly' ciphers, or design such ciphers. And then find parameters that lead to an optimal performance vs. security trade-off. In our approach, we shall aim for software-friendly constructions, i.e., where a system of multivariate polynomials can be defined over a finite field of characteristic 2, and a byte, word, or quadword size. In other words, constructions defined over finite fields GF $\left(2^{i \cdot 8}\right)$.

Approach

To achieve our objective, we need to define a cryptographic cipher as a system of equations, and then propose how to obfuscate it. The construction of such cipher needs to meet several requirements. A cipher cannot just be a random system of multivariate equations. It needs to be an *invertible* and *secure* system. In general, random systems are not invertible, or their inverse is hard to compute and too complex for practical purposes. Therefore, we first search in the literature what kind of constructions exist. We do this in Section 11.1.3. Of each of these constructions, it is important to understand its properties; what makes them strong or weak; what makes it efficient or what makes that these systems can be (trapdoor) inverted.
Subsequently, we analyse the practical implication of defining and implementing such systems. The practical impact with respect to size and performance. This depends on the field over which the system of equations is defined, the dimension of the multivariate map, and the degree of the multivariate equations. This investigation is executed in Section 11.1.4
The third step of our research activity is then to perform a security analysis. Given a selection of parameters, we need to know if constructions can meet our desired level of robustness. In Section 11.1.5, perform a thourough security analysis, which leads us to some important conclusions 11.1.6.

11.1.3 Multivariate cipher constructions

Multivariate public key cryptosystems (MPKCs) are a family of cryptosystems based on multivariate equations. Unlike conventional public key cryptosystems such as RSA, they are expected secure against quantum computers and their operation are expected to be quicker since they do not need exponentiation operations. The downside however is the size of their public key.
MPKC was introduced by **Matsumoto and Imai** in 1983 [52]. Their MI cryptosystem [57] builds around the function $F(X) = X^{q^\theta+1}$, in the finite field GF (q^n). $F(X)$ is a quadratic mapping and can be described as n quadratic polynomials over $(x_1, \ldots, x_n)$. The inverse of F can easily be derived: X^h, with $h = (q^\theta + 1)^{-1} \mod q^n - 1$. The public key is defined as $P = T \circ \pi^{-1} \circ F \circ \pi \circ S$, where π is an isomorphism between GF $(q)^n$ and GF (q^n), and S and T are two linear bijective maps over GF $(q)^n$.
The MI cryptosystem was successfully cryptanalyzed by Patarin [63] in 1995, where bilinear relations over GF (q) can be constructed. These bilinear relations can be constructed through differential cryptanalysis (derivate functions). When n bilinear relations using $(n+1)^2$ pairs (plaintext,ciphertext) are found, then we can derive a linear system which has an n-dimensional kernel that can be found. The MI scheme "B" was cryptanalyzed by Patarin and Youssef [72].
At Eurocrypt 2006, Faugère and Perret describe a Gröbner basis algorithm to solve the IP problem when B is a set of polynomials defined over a small n of variables in an extension F [39]. Their algorithm is very efficient when the system of polynomials B is *random* and has small degree terms such as in the authentication scheme proposed by Patarin [65] and some parameters of the traitor tracing scheme of Billet and Gilbert [7]. For larger parameters proposed by Billet and Gilbert, or for the parameters of SFLASH, the algorithm does not work. In the case of SFLASH, the system of polynomials is too sparse. At Eurocrypt 2008, Fouque *et al.* present a polynomial time attack to recover these values for SFLASH and the second parameter proposed by Billet and Gilbert [44].
SFLASH was build around the C^* scheme of Matsutomo and Imai. The main idea is to remove some equations from the public key (hence, not enough equations can be found to deploy the Patarin attack). This is also called the C^{*-} scheme. The ground field is $K = \text{GF}\left(2^7\right)$. The secret key comprises two invertible linear transformations over K^{37} and an isomorphism $\phi : K^{37} \to L$, where L is an extension of degree 37 over K defined by the primitive polynomial $x^{37} + x^{12} + x^{10} + x^2 + 1$.
Dubois *et al.* presented an attack on SFLASH [34, 33], based on Patarin's attack [63]. However, the attacks were not able to recover the secret key, as they rely on Patarin's attack which is only able to invert the public key.

Rainbow is a signature scheme, where the public key consists of 27 multivariate equations in 33 unknowns over the finite field GF (2^8) [30]. However, Rainbow has been broken by Billet and Gilbert [8].
PMI$^+$ is a doubly perturbed C^* scheme [29]. The public key is a multivariate quadratic system of 98 equations in 84 unknowns defined over GF (2). It comprises an exponentiation function $F : x \to x^{2^4+1}$ defined over a finite field GF (2^{84}), and two invertible linear transformations over GF (2^{84}) and GF (2^{98}) respectively. Additionally, 14 extra quadratic polynomials in the 84 unknowns defined over GF (2) are chosen.
The stream cipher **QUAD** takes advantage of specific characteristics of multivariate systems of equations in order to provide some provable security properties [5].
Several **hash functions** based on multivariate constructions have been presented. These include the hash functions of Billet *et al.* [10], and Ding and Yang [31]. The latter presented three approaches: “dense” cubic random polynomials, “sparse” cubic random polynomials, and the composition of 2 quadratic random systems (i.e., degree 4 polynomials).

A(NA)* construction

The pioneering papers of Matsumoto and Imai lead to other constructions that have been proposed based on different sets of quadratic multivariate systems, including *one-round schemes* [65]. The one-round construction comprises quadratic S-boxes (over GF (2)). In their paper, Patarin and Goubin present four possible attack strategies on this construction [65, 63].
To strengthen these schemes, Patarin and Goubin extended the idea to a two-round family of schemes [65]. The public key of such systems, which is given by polynomials of degree four, is obtained by composing the public polynomials of two different instances of one-round schemes. The “2R” construction comprises non-bijective S-boxes, and was cryptanalyzed by Biham [6]. Related literature includes the SASAS paper by Biryukov and Shamir [11], a generic attack in GF(2) [12], and generalized attacks on systems that consist of two rounds of quadratic systems of equations [71].

Hidden Field Equations (HFE)

HFE is a family of cryptosystems introduced by Patarin [64]. The cryptosystem is defined as polynomials over finite fields; the recommendation is to implement the cipher over GF (2^n) for $127 \leq n$ with a univariant polynomial of degree $25 \leq d \leq 33$ and d to be odd. While public key operations (that is, encryption and signature verification) are quite fast, private key operations (that is, decryption and signature generation) are much slower. Another drawback is the very large public key size of about 100kb for $n = 129$. The key size grows with a complexity of $O(n^3)$ and is for example around 1Mb for $n = 257$. Cryptanalysis results have been presented that threaten the security of HFE. Nevertheless, it is possible to define variants on HFE for which the attacks no longer apply. One of these variations (called “HFE-”) involves hiding some of the public equations, another (called “HFEv”) introduces some more variables, the so-called “vinegar variables”.

The IP problem (Isomorphism of Polynomials) has been used to construct public key schemes. Via the Arthur Merline Games proof systems, it can be shown that the IP rpbolem is not NP-complete. Yet, the more general Morphism of Polynomials (MP) problem, which does not restrict to invertible components (more particular, non invertible matrices can be used), is proved to be NP-complete (See Micheal Garey and David S. Johnson – computers and intractability, 1979 [45]). IP and MP have deep links with famous problems such as the Isomorphism of Graphs problem and the problem of fast multiplication of $n \times n$ matrices. The problem of deciding wether a polynomial isomorphism exists between two sets of equations is *not* NP-complete, but solving IP is at least as difficult as the Graph Isomorphism problem.

Polly Cracker

In 1994, Fellows and Koblitz proposed a framework for the design of public-key cryptosystems based on multivariate algebras [43]. The polynomials generating the public ideal are derived from combinatorial or algebraic NP problems; and the schemes are bound to be sparse: the reasons are to render the linear algebra attack exponential time, and to allow for a reasonable-size public key. The public key are the polynomials $\{f_i\}$, generating an ideal $I = (f_i)$ in a ring; the private key is a Gröbner basis G of an ideal J containing I. The main illustration of such a system was the Polly Cracker cryptosystem, which is a special case in which J is a maximal ideal (a root of $f_i(X) = 0$).
In 1998, Grant *et al.* presented a public key cryptosystem, where Alice's public key comprises a set of sparse trapdoor random multivariate polynomials [50]. The trapdoor information are the roots of the generated public polynomials; the security is related to the difficulty of finding these roots.
In 2002, Steinwandt *et al.* presented several attacks on Polly Cracker; Grant's public key system can be seen as a special instance to which the idea of this attack can be applied [66].
The fallacy of the idea is that you don't need to find the Gröbner basis, or a normal form; you only need to find the g_i such that $c - \sum g_i f_i \in V$. In general, to attack Polly Cracker algorithms:

- If the f_i are dense, just use linear algebra.
- If the f_i are sparse, guess the support of the computation and use sparse linear algebra.

It was shown by Hofheinz and Steinwandt that guessing is easy provided that the f_i (and anything you can derive easily) have and least 3 monomials each [51]. The only case that seems to escape this, is the case of a binomial ideal, i.e., an ideal generated by binomials $X^\alpha - X^\beta$ [16]. Since those ideals are very special inside Gröbner bases theory, it might be that this case is more secure than its predecessors. In particular, this construction can be defined as a lattice Polly Cracker algorithm, yielding results related to NP hard lattice based problems. However, the interest in this field may stop there, since other lattice-based constructions may result in more secure and more efficient algorithms.

11.1.4 Implementation

To define a multivariate white-box implementation, we need to (1) define a multivariate equation system with appropriate parameters and (2) obfuscate the system. In the Section 11.1.3 above, we performed an analysis of cryptographic ciphers based on multivariate systems to figure out what constructions may be appropriate. In Section 11.1.5, we perform a security analysis on Multivariate WBC constructions to figure out which parameters are appropriate. I.e., over what (Galois) fields, dimension of the multivariate mapping, and what degree of the multivariate mapping is acceptable. This needs to define some lower bounds. In this section, we perform an analysis w.r.t. size and performance. These three approaches (constructions, size, security) need to lead to our conclusions for multivariate white-box cryptography.

Size and performance estimations

Table 13 presents an overview of the size and performance numbers that we would expect when an arbitrary cipher comprising of 4 systems of equations (each representing a round) would be implemented. This includes a time-memory trade-off where the polynomial terms are pre-computed to speed up performance in return for memory usage.
The two last columns of the table present performance figures. These are theoretical numbers; in reality, performance figures heavily depend on the platform and implementation. Cache size, CPU speed, and CPU features such as superscalar CPU's or SIMD instructions heavily influence processing speed. But at least, they provide an indication that can be used as reference for upper bounds and to help decide which ones to implement for further analysis. As a rule of thumb, we have assumed that additions take one cycle, while multiplications take 4 cycles. The latter is under the assumption that Galois multiplications are implemented as a combination of power and

Table 13: Expected implementation size of systems of equations and expected performance

Base field	deg	Size	Additions	Multiplications	ms	kbit /sec
GF $(2^8)^{16}$	4	302.81 kB	310,016	321,024	1.668	76.734
GF $(2^8)^{16}$	6	4.55 MB	4,775,168	5,158,272	26.508	4.829
GF $(2^8)^{16}$	8	44.89 MB	47,070,080	53,530,752	271.528	0.471
GF $(2^8)^{16}$	9	124.69 MB	130,750,336	152,901,056	770.311	0.166
GF $(2^8)^{16}$	12	1.81 GB	1,946,992,256	2,486,914,560	12272.021	0.010
GF $(2^{16})^8$	4	30.93 kB	15,808	17,600	0.090	1426.149
GF $(2^{16})^8$	6	187.69 kB	96,064	118,976	0.591	216.480
GF $(2^{16})^8$	8	804.37 kB	411,808	572,000	2.771	46.191
GF $(2^{16})^8$	9	1.48 MB	777,888	1,144,000	5.477	23.371
GF $(2^{16})^8$	12	7.69 MB	4,031,008	6,987,136	32.420	3.948
GF $(2^{32})^4$	4	4.38 kB	1,104	1,456	0.007	17943.547
GF $(2^{32})^4$	6	13.12 kB	3,344	5,376	0.025	5061.000
GF $(2^{32})^4$	8	30.94 kB	7,904	15,312	0.070	1835.983
GF $(2^{32})^4$	9	44.69 kB	11,424	24,112	0.108	1181.661
GF $(2^{32})^4$	12	113.75 kB	29,104	77,168	0.336	380.945

log tables. In the case of GF (2^8), performance on ARM platforms (as targeted in ASPIRE) may in reality be much faster because the XOR operations can be executed in parallell (since the target ARM processor is a superscalar CPU), while on large fields such as GF (2^{32}) the performance may be slower because Galois multiplications become more complex (power and log table approach becomes inappropriate). From cycles to ms, we assume a 3 Ghz clock speed.

Empirical results

To get a grasp on more exact numbers, we have implemented a dummy cipher, comprising of four consecutive rounds. Each round is defined as a multivariate system of equations over GF (2^q) that we generated randomly, by generating a number of random terms and coefficients for each polynomial.
C-code for each of the random ciphers has been generated and compiled with GCC with the -O2 optimization flag enabled. Since GCC seems to fail to compile routines that exceed close to or more than 5000 lines of code, we have split the code into a number of routines such that each routine contains approximately 3000 lines of code. The compiled code has been benchmarked on an Intel Dualcore 3Ghz CPU test platform. The presented performance measures are the average of 100,000 tests on systems of random equations.
Table 14 contains the results of our tests.

Performance and Size of Concrete Constructions

The numbers presented in Table 14 relate to systems with *random* generated equations, with a density of 1. In real-world cryptographic use-cases, there are a number of constraints that need to be fulfilled. The most important one is that the system needs to be (trapdoor) invertible. In general, this is not the case for randomly generated multivariate maps. The same holds for the obfuscating annihilating mappings that are used: those too need to be invertible.
There are a number of ways in which invertible multivariate maps can be created. In cryptography, a popular approach is via Feistel constructions. Alternatives include Lai-Massey, SASAS-alike constructions, MI-based constructions, constructions based on T-functions, or HFE alike constructions. We refer to Section 11.1.3 for more details on these constructions. We implemented and randomized several of these constructions. Our observation is that the obfuscated versions (where

Table 14: Empirical results of fixed-key 128-bit multivariate systems.

Base field	deg	#terms	real #terms	LOC	binary size	ms
$GF(2^{32})^4$	6	5000	161	2,082	108.80 kB	0.173
$GF(2^{32})^4$	8	5000	299	3,596	204.85 kB	0.306
$GF(2^{32})^4$	9	5000	374	4,453	260.85 kB	0.382
$GF(2^{32})^4$	12	1000	521	5,393	312.85 kB	0.454
$GF(2^{32})^4$	12	10000	627	7,605	440.90 kB	0.673
$GF(2^{16})^8$	6	1000	1,720	22,947	1.39 MB	0.341
$GF(2^{16})^8$	6	5000	1,984	45,757	2.79 MB	0.683
$GF(2^{16})^8$	6	10000	2,021	51,742	3.16 MB	0.784
$GF(2^{16})^8$	8	1000	3,598	29,990	1.81 MB	0.466
$GF(2^{16})^8$	8	5000	5,572	88,740	5.56 MB	1.690
$GF(2^{16})^8$	8	10000	6,150	121,604	7.64 MB	2.450
$GF(2^{16})^8$	9	1000	4,440	32,544	1.94 MB	0.512
$GF(2^{16})^8$	9	5000	8,298	106,738	6.67 MB	2.128
$GF(2^{16})^8$	9	10000	9,567	157,651	9.95 MB	3.134
$GF(2^{16})^8$	12	1000	6,164	41,764	2.35 MB	0.634
$GF(2^{16})^8$	12	5000	17,263	151,995	9.18 MB	3.091
$GF(2^{16})^8$	12	10000	23345	252,806	15.54 MB	5.154

subsequent rounds are obfuscated with annihilating systems of random multivariate equations) tends to have a density close to 1. Therefore, the performance and size results for specific constructions are comparible with the performance and size results presented in Table 13, for the case of $GF(2^{16})^8$. We did not perform a comparable analysis for the case of $GF(2^8)$ or $GF(2^{32})$ due to lack of time, and chose to focus on $GF(2^{16})$ as it seemed most appropriate w.r.t. size and performance.

11.1.5 Security Analysis

The decomposition of random system of equations over $GF(2)$ is NP-hard [26]. Moreover, in contrast to Discrete Logarithm or RSA-based systems, no polynomial-time quantum algorithm is known.
This intractability theorem however, does not apply to multivariate equation systems that are the composition of specific structures, which is what we aim to construct in our approach. In this subsection, we make a security analysis of generic MWBC constructions, mainly based on academic papers from the domain of cryptanalysis of multivariate systems and algebraic analysis techniques. The goal is to find adequate parameters for which the security level can be considered adequate.
Analysis techniques include Rank attacks, Distinghuiser attacks, attacks tailored to specific constructions such as SASAS [11], Gröbner basis attacks, or Derivative Attacks. We have been looking into all of these attacks; given our experience, we will briefly introduce Gröbner basis attacks, and more extensively discusss Derivative Attacks.

Gröbner basis

In his 1965 thesis, Buchberger developed the theory of Gröbner bases which allow computations in multivariate polynomial rings analogous to those we are used to in single variable polynomial rings. This theory can be seen as a generalization of Gaussian elimination or of integer programming. Hence, Gröbner bases are a fundamental tool of commutative algebra.

Defininition 3 (Buchberger 1965 [15]) *$G = \{g_1, \ldots, g_t\} \subset K[x_1, \ldots, x_n]$ is a Gröbner basis of a polynomial ideal I, if*

$$\forall \in I, \exists g \in G \text{ s.t. } LM(g) \text{ divides } LM(f)\,,$$

where LM denotes the leading terms of a polynomial.

Note that this depends on the monomial ordering.
Gröbner bases are a useful tool in polynomial algebra. In particular to solve the problem to find roots for systems of equations over polynomial algebras. The problem where given $f_i(x_1, \ldots, x_n) \in K[x_1, \ldots, x_n]$, find $z = (z1, \ldots, z_n) \in K^n$ such that $f_i(z) = 0$.
The challenge is to efficiently compute Gröbner bases, since existing algorithms are of exponential complexity in time/space. Several Gröbner bases exist such as Lexicographic Gröbner basis (LEX) or degree reverse lexicographical Gröbner bases (DRL). Faugère *et al.* showed that computing LEX is much slower than computing DRL [38].
Several algorithms exist to compute a DRL Gröbner basis. First Buchberger's algorithm, which is based on the Buchberger normal form algorithm [15], which is a Gaussian-like linear algebra reduction. Given a finite set F and an element f, it returns the normal form g of f w.r.t. F and a strong Gröbner representation of $f - g$ in terms of F.
Improved algorithms are the F4 [35], F5 [36] and Fast FGLM [37] algorithms of Faugère *et al.* Unfortunately, time did not permit to investigate F5 in sufficient detail to figure out its limitations.

Derivative Attacks

Derivative attacks are generic attacks on the functional decomposition problem (FDP). These attacks are in particular interest for us as it directly targets the approach we conceive: it are attacks that can be used to attempt to distinguish the *obfuscating multivariate map* from our multivariate cryptographic cipher.
Surprisingly, the very first technique addressing multivariate functional decomposition was only in 1999 by Ye *et al.* who proposed an efficient algorithm for decomposing a set of n multivariate polynomials of degree four into two sets of n quadratic polynomials [71]. This technique has been used to attack the "2R" cryptosystem and exploits techniques used in linear algebra. However, it is limited to the special case where the number of variables equals the number of polynomials (which is the case in our approach), and only for polynomials of degree four.
Faugère and Perret proposed a new algorithm allowing to decompose polynomials of the same arbitrary degree [40]. This was further improved by the same authors by using high order partial derivatives [41]. This new approach results in a simple, natural way of decomposing multivariate polynomials. Finally, this is further extended by Faugère *et al.* [42] for the decomposition of generic multivariate polynomials. The complexity of the attack depends on the degree of the polynomials, and on the ratio between the number of variables (n) and the number of polynomials (u). We will now dive into the details of these attacks, since it is important to understand how it works and up to what extent it can be applied.

Consider the multivariate mappings $f = (f_1, \ldots, f_u)$ and $g = (g_1, \ldots, g_n))$ of degree d_f, d_g respectively and its composition

$$h = (h_1, \ldots, h_u) = (f_1(g_1, \ldots, g_n), \ldots, f_u(g_1, \ldots, g_n)) \in K[x_1, \ldots, x_n]^u\,.$$

We shall say that (f, g) is a (d_f, d_g) decomposition of h.
The FDP challenge is to find a non-trivial decomposition of a given multivariate map h. If one solution exists, then this solution can not be unique; indeed, for any bijective linear $A \in GL_n(K)$, it holds that $h = (f \circ A^{-1}) \circ (A \circ g)$. Therefore, it suffices to compute the space span$\{g_i\}$, since any basis for this space is a candidate solution for g. Once such a system g has been found, it is straightforward to compute the corresponding f.

Quadratic Derivative Attacks

The attack presented by Ye *et al.* was designed to attack the $2R^-$ schemes, and is limited to decompose degree 4 mappings (combination of quadratic mappings), where the number of polynomials equals the number of variables ($u = n$) [71]. The attack was extended by Faugère and Perret [40] to polynomials of arbitrary degree and the case where $u \neq n$. Their algorithm allows to decompose polynomials of degree four in $O(n^{12})$ if $n/u < 1/2$ and $O(n^9)$ if $u = n$. The algorithm is divided in two parts. First, to recover the vector space $L(g) = \text{Span}(g_1, \ldots, g_n)$. This linear span will be recovered from the DRL Gröbner bases of suitable ideals. Secondly, deduce a decomposition (f, g) of h from $L(g)$.
The first step – recovering the linear span – is the most important step. In the $(2, 2)$-case, observe that

$$\frac{\partial h_i}{\partial x_j} = \sum_{k,l} f_{k,l}^{(i)} \left(\frac{\partial g_k}{\partial x_j} g_l + \frac{\partial g_l}{\partial x_j} g_k \right),$$

where the partial derivatives $\frac{\partial g_k}{\partial x_j}$ are of degree one. Hence,

$$\partial I_h = \left\langle \frac{\partial h_i}{\partial x_j} \right\rangle \subseteq \langle x_k g_l \rangle .$$

This ideal usually provides enough information for recovering a basis of $L(g)$.

Higher Order Derivative Attacks

Let us observe the $(3, 2)$ case; that is, where the degree of f is 3, and the degree of g is 2. Then $h_i = f_i(g_1, \ldots, g_n) = \sum_{k,l,m} f_{k,l,m}^{(i)} g_k g_l g_m$, and hence

$$\frac{\partial h_i}{\partial x_j} = \sum_{klm} f_{klm}^{(i)} \left[\frac{\partial g_k}{\partial x_j} g_l g_m + \frac{\partial g_l}{\partial x_j} g_k g_m + \frac{\partial g_m}{\partial x_j} g_k g_l \right] .$$

By considering the second order partial derivatives, we get

$$\begin{aligned} \frac{\partial^2 h_i}{\partial x_j x_r} = \sum_{klm} f_{klm}^{(i)} \quad & \left[\frac{\partial g_k}{\partial x_j} \frac{\partial g_l}{\partial x_r} g_m + \frac{\partial g_l}{\partial x_j} \frac{\partial g_m}{\partial x_r} g_k + \frac{\partial g_k}{\partial x_j} \frac{\partial g_m}{\partial x_r} g_l + \right. \\ & \frac{\partial g_k}{\partial x_r} \frac{\partial g_l}{\partial x_j} g_m + \frac{\partial g_l}{\partial x_r} \frac{\partial g_m}{\partial x_j} g_k + \frac{\partial g_k}{\partial x_r} \frac{\partial g_m}{\partial x_j} g_l + \\ & \left. \frac{\partial^2 g_k}{\partial x_j \partial x_r} g_l g_m + \frac{\partial^2 g_l}{\partial x_j \partial x_r} g_k g_m + \frac{\partial^2 g_m}{\partial x_j \partial x_r} g_k g_l \right] . \end{aligned}$$

Define the ideal $\partial^2 I_h = \left\langle \frac{\partial^2 h_i}{\partial x_j \partial x_r} \right\rangle$, which will provide enough information to recover $L(g)$. Observe that each $x_k x_l g_m$ can be expressed as a linear combination of polynomials in the ideal. Hence, computing a reduced DRL Gröbner bases leads to a bases for $L(g)$: let G be the Gröbner bases of $\partial^{(d_f-1)} I_h$, and $B_g = \{g \in G | \deg(g) = d_g\}$, then $L(g) \subseteq \text{span}_K\{B_g\}$. The equality holds if its dimension is n. If the rank of the matrix that represents the ideal is not sufficiently large, we can consider a bigger vector space by *stretching* the matrix.
From $L(g) = \text{span}(g_1, \ldots, g_n)$, a candidate decomposed multivariate polynomial g is any non-zero $g \in L(g)$. The corresponding f such that $h = f \circ g$ can then be recovered by solving a linear system of equations.

This approach can be generalized for each $d_f \geq 2$, by defining the ideal

$$\partial^{(d_f-1)} I_h = \left\langle \frac{\partial^{(d_f-1)} h_i}{\partial x_{j_1} \ldots \partial x_{j_{(d_f-1)}}} \right\rangle .$$

Each polynomial of this ideal will be of the form

$$\sum_{l=1}^{n} H_l(x_1, \ldots, x_n) g_l \, ,$$

where each H_f is 0 or a polynomial of degree $(d_g - 1)(d_f - 1)$. Solving the corresponding matrix with a DRL Gröbner bases will then lead to $L(g)$, and ultimately to the decompostion (f, g). If the algorithm works correctly, it will return a unique decomposition with respect to the "normal form" of a multivariate decomposition.

Limitations.

For the algorithm to work, there needs to be sufficient data to run the Gröbner reduction algorithm. For small values of n, the algorithm often fails (independently of the fact that a decomposition exists or not). For example, let $h = (h_1, \ldots, h_n)$ be generic polynomials of degree four. The algorithm will always return `Fail` if

$$|M(3)| - 2 \cdot n < n^2 \, ,$$

where $M(\delta)$ is the set of monomials of degree $\delta \geq 0$ in $x_1, \ldots, x_n$, and $| \cdot |$ denotes the cardinality of the set. In a $(3, 2)$ decomposition $(u = n)$, the algorithm will always return `Fail` if $n < 6$.

11.1.6 Conclusion

In our research, we have focused on

- The analysis of existing constructions to figure out which constructions are appropriate.
- Subsequently, we have investigated upper bounds on size and performance from both an exercises on paper, as weel as by implementing some empirical tests on random system of equations. On some selected concrete implementations (based on Feistel constructions), we have observed that these constructions tend to be close to the estimated upper bounds.
- Last but not least, to figure out which parameters (size of the multivariate map, degree of the equations) are appropriate, we have conducted a thorough security analysis.

From this research, we conclude that this direction of constructing white-box implementations based on multivariate equations is indeed viable. But only for ciphers with small block size and for which a low performance and large implementation size is acceptable. Indeed, from the security analysis it becomes clear that the degree of the multivariate equations needs to be sufficiently high to ensure that derivate attacks cannot be applied. For such equations, as soon as the block size becomes too large; the performance and size impact becomes quickly unacceptable.

11.2 White-Box Cryptography based on Fully Homomorphic Encryption

Section authors:
Jerome d'Annoville (GTO)

The purpose of this section is to explains why Gemalto could not work on the Fully Homomorphic Encryption (FHE) as it was initially planned at the beginning of the project. A replacement work is described in the next section.

11.2.1 Work description and motivation

As part of the description of the task T2.2 Gemalto committed to the use of constructions of FHE in WBC solutions. FHE is used to evaluate arbitrary functions over encrypted data without decrypting the data. The motivation of working on this topic is ultimately to deploy software protection techniques for mobile devices and in this sense FHE schemes are not computationally practical for use on handheld and mobile devices. Various research works (cf. [47], [67], [23], and [48]) have reported the hardware computational power that would be required to keep an implementation into an acceptable processing timescale. Somewhat homomorphic encryption schemes (SWHM) provide a compromise. With SWHM techniques, it is not possible to evaluate arbitrary functions, and SWHM functions need to be explicitly written to perform a very specific type of operation (e.g. standard deviation, average of n terms, etc) but still, the research work was planned to explore this lead for the context of mobile devices.
It happens that this work did not run through for the various reasons explained hereafter:

- Personal issue and backup plan

  The main contributor for this research task resigned in 2013 and no other person was technically able to work on the topic in Gemalto Europe due to the specific skill that is required. The backup plan was to hire a PhD to work full time on WBC to do the actual research work. Part of the result would be provided as Gemalto ASPIRE contribution in T2.2 and an external sub-contractor that has the required background would have steered and technically managed the researcher. With this new plan the deliverable milestone would need to be shifted in M24 for early results and M30 for final results while contribution was expected at M18 in the DoW. Then the process to hire the researcher has been initiated at its own speed but it has been blocked by the management and eventually rejected in Q2 because an interrelated project within Gemalto has been started in the meantime. This decision should be considered in the light of the availability of new technologies such as the one described below.

- Google Host Card Emulation

  Google Host Card Emulation (HCE) is part of Android 4.4 available since Q4, 2013. Devices running this system release are commonly available since Q1, 2014. HCE brings a new behavior: when a device has an embedded Near Field Communication (NFC) controller in the device and when the device is tapped on a card reader or a Point Of Sale the system may route commands sent by a physical card reader/POS to an Android service. Until Android Jelly Bean (4.3) the Contactless frontend (CLF) of the device directly send commands to the SIM card chip exclusively. This new feature weakens the NFC-SIM Market of Gemalto with clients that tend to question the secure hardware approach, which force the company to accelerate the development of pure software security on mobile devices. In a way it can be seen as if the main motivation that initiates the ASPIRE project has been endorsed by a market fact just few months after the project has started.

- Management arbitration

  A pure software library product is designed in Gemalto to support the implementation of HCE. It brings another way to secure the cryptography code in Android applications. This technology is based on WBC to secure the keys used to secure the library. The WBC layer is licensed to Gemalto because there is no time to develop this layer internally and effort must be put on higher layers of the library. The management has arbitrated in Q2 that the WBC layer has to be bought because designing and implementing it within the company would delay the availability of the solution in such way that it would compromise its commercial success. This decision impacts ASPIRE because the research task as proposed in ASPIRE has been re-assessed in light of the decision to buy WBC layer license. It has been decided not to duplicate effort in various directions and even if long term research on the SWHM approach would be more profitable for the company it conflicts with the decision of buying the technology. There is no time left for the internal research activity and an already available implementation is preferred otherwise the marketing window would be missed. The negative consequence is the withdrawal of Gemalto on this WBC contribution in ASPIRE. The other consequence is that the library currently developed could be used in ASPIRE to provide a cryptography library secured by pure software. This alternative, the Diversified Crypto Library, is described in the next section.

11.3 Diversified Crypto Library

Section authors:
Jerome d'Annoville (GTO)

Purpose of this protection is to provide a library that combines the WBC techniques together with a server based diversification technique. Features provided by the library are those of a classical cryptography interface except that secret and private keys are kept within the library. With this protection, the application downloaded from Google Play only embeds the minimal library component called the initial crypto library. A library bootstrap module is part of the initial crypto library as shown in figure 19. When the application is launched for the first time, the library bootstrap module initiates a provisioning request to the Crypto Library Provisioning Server. A secure communication is set with the Provisioning Server to download the library. This crypto library is a personalized library for a dedicated Android mobile device. Provisioning Server generates unique keys and embeds these keys to the personalized library binary.

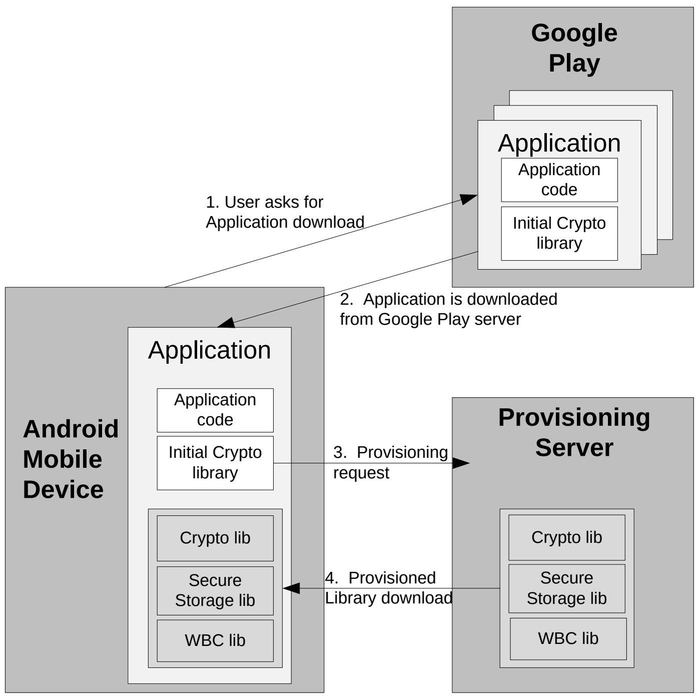


Figure 19: Application and crypto library downloads

Another design would be to have the Application together with a complete crypto library integrated in the Android package to avoid the provisioning step when the application is launched for the first time. Because the crypto library binary is generated on purpose with the appropriate keys for both a dedicated application and a dedicated device there is no other option than to have some mobile code that has to be downloaded after the application has been installed on the device. The library bootstrap module provides a way to provision and download a personalized key

material binary of the crypto library to a mobile device. It contains minimum set of crypto APIs, the public key of the provisioning server and the necessary implementation of cryptographic algorithms to initiate the secure connection with the Provisioning Server.
When the provisioning of the library is finished the crypto library provides application with a full crypto API. The crypto library can store securely data thanks to a WBC library that is provisioned with root keys. Based on these crypto and secure storage services the application can start its application data provisioning.
There are two root keys. A unique Key Encryption Key (KEK) is used to protect the secure storage containers. Application keys are stored in secure storage containers. Another key, the Key Masking Key (KMK) is used to obfuscate application keys. Both KEK and KMK keys are generated on the provisioning server and embedded into one occurrence of the WBC library. These two keys are unique for a library occurrence for a device. Again, the full crypto library with its services is not part of the Android package (.apk) when downloaded from Google Play. When the software application is first launched after download, it communicates with the provisioning server to retrieve the customised library that is installed locally on the device.
Features can be summarized as follows:

- The library provides standard encryption algorithms with application keys embedded in the library.
- The application keys are protected by a secure storage service
- Some specific keys - the root keys- are managed internally in the secure storage service of library and protect the application keys.
- These specific keys are protected themselves by WBC.
- The key generation, diversification and key management are moved out of the handset.

More details on the architecture of the library will be given during the first project review.
This Diversified Crypto protection better fits in the WP3 of the ASPIRE project because it is an online protection that relies on servers. Nevertheless, it is described in this deliverable for consistency as it is a replacement work. It will be described in WP3 in the project amendment.

12 Applied WBC

Section authors:
Brecht Wyseur, Patrick Hachemane (NAGRA)

Task 2.2 comprises three large activities: (1) the R&D on new white-box schemes with provably secure properties as presented in Section 11, (2) the implementation of a tool to generate and manage white-box code as presented in Section 10, and (3) R&D on weaker algorithms which lower overhead than the provably secure ones.
So far, our activities mainly focused on (1) and (2). In the remainder of the task execution – year 2 of the ASPIRE project – we will focus on (3) and improve the WBTA tool to accomodate the applied WBC schemes. We envision two steps to complete this activity, which we brielfy discuss in this section.

12.1 Dynamic white-box implementations

So far, the state of the art in WBC comprises fixed-key white-box implementations – where the cryptographic key is hard-coded into the implementation. These are of limited use in practice. Dynamic WBC allow to instantiate a (pre-generated) white-box implementation after it has been deployed in the field. This is similar to traditional use-cases where for example an AES implementation is compiled into an application, and then later on, the AES key is provided to the application when needed.
Similarly, dynamic white-box implementations are generated at compile-time and linked into the target application. This generation process comes without a cryptographic key (in constract to fixed-key white-box implementations). The key is not known at the time of generation, and the implementation needs to be instantiatable with arbitrary keys. Obviously, the key cannot be provided 'in the clear' (unprotected) to the application when needed. Instead, the key needs to be transformed into a specific form that protects it from analysis and that allows the white-box implementation to process it in a secure way. We denote this transformation as *key obfuscation*.
This requires additional support at server-side: to *obfuscate* cryptographic keys in such a way that the client-side white-box implementation is able to process it. As presented in Section 10.6, we have been extending the WBTA tool to support the generation of relevant sever-side code. The remainder of this activity will comprise the design and implementation of such white-box schemes.

12.2 Time-limited white-box implementations

Another direction that we are investigating is towards *time-limited* implementations. The idea is to generate white-box implementations that are tuned towards performance. This would mean that they are less secure, but we would compensate this by renewing the implementation frequently. The renewal process means that we would generate new instances at server-side, and then exploit renewability techniques (as investigated in Work Package 3) to deliver and deploy these new instances at server-side.
In contrast to dynamic white-box implementations, these time-limited implementations would be fixed-key implementations. In general, fixed-key implementations achieve a significantly better performance versus security trade-off. Instantiating a new key would require to renew the complete instance. The remainder of this activity will comprise the design and implementation of such white-box schemes.

List of abbreviations

ACTC	Aspire Compiler Tool Chain.
ADSS	Aspire Decision Support System.
AES	Advanced Encryption Standard.
CBC	Cipher Block Chaining.
CLF	Contactless Front End.
DES	Data Encryption Standard.
ECB	Electronic Code Book.
FDB	Functional Decomposition Problem.
FHE	Fully Homorphic Encryption.
GCD	Greatest common divisor.
HCE	Host Card Emulation.
IV	Initialization Vector.
MWBC	Multivariate White-Box Crypgraphy.
POS	Point Of Sale.
RNC	Residue Number Coding.
SIM	Subscriber Identity Module.
SWHM	Somewhat Homomorphic.
VM	Virtual Machine.
WB	White-Box.
WBC	WB Cryptogrraphy.
WBT	WBC Tool.
WBTA	WBT for Aspire.

References

[1] Paul Anderson and Tim Teitelbaum. Software inspection using codesurfer. In *Proceeding of the First Workshop on Inspection in Software Engineering*, Paris, France, july 2001.

[2] ASPIRE Consortium. ASPIRE Compiler Tool Chain and Decision Support System Design - Document D5.01b, 2014.

[3] ASPIRE Consortium. White-box tool - Reference Guide - Document D2.04, 2014.

[4] ASPIRE Consortium. White-box tool for Aspire - Reference Guide - Document D2.04b, 2014.

[5] Côme Berbain, Henri Gilbert, and Jacques Patarin. QUAD: A multivariate stream cipher with provable security. *J. Symb. Comput.*, 44(12):1703–1723, 2009.

[6] Eli Biham. Cryptanalysis of Patarin's 2-Round Public Key System with S Boxes (2R). In *Advances in Cryptology - EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 408–416. Springer-Verlag, 2000.

[7] Olivier Billet and Henri Gilbert. A Traceable Block Cipher. In *Advances in Cryptology - ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, 2003.

[8] Olivier Billet and Henri Gilbert. Cryptanalysis of rainbow. In *Security and Cryptography for Networks Conference – SCN 2006*, volume 4116 of *Lecture Notes in Computer Science*, pages 336–347. Springer, 2006.

[9] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a White Box AES Implementation. In *Proceedings of the 11th International Workshop on Selected Areas in Cryptography (SAC 2004)*, volume 3357 of *Lecture Notes in Computer Science*, pages 227–240. Springer-Verlag, 2004.

[10] Olivier Billet, Matthew J. B. Robshaw, and Thomas Peyrin. On building hash functions from multivariate quadratic equations. In *Information Security and Privacy Conference – ACISP 2007*, volume 4586 of *Lecture Notes in Computer Science*, pages 82–95. Springer, 2007.

[11] Alex Biryukov and Adi Shamir. Structural cryptanalysis of sasas. In *Advances in Cryptology - EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 394–405. Springer, 2001.

[12] Julia Borghoff, Lars Knudsen, Gregor Leander, and Soren S. Thomsen. Cryptanalysis of PRESENT-like block ciphers with key dependent components. 2010.

[13] Julien Bringer, Hervé Chabanne, and Emmanuelle Dottax. Perturbing and Protecting a Traceable Block Cipher. In *Proceedings of the 10th Communications and Multimedia Security (CMS 2006)*, volume 4237 of *Lecture Notes in Computer Science*, pages 109–119. Springer-Verlag, 2006.

[14] Julien Bringer, Herve Chabanne, and Emmanuelle Dottax. White box cryptography: Another attempt. Cryptology ePrint Archive, Report 2006/468, 2006. `http://eprint.iacr.org/`.

[15] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, 1965.

[16] Massimo Caboara, Fabrizio Caruso, and Carlo Traverso. Gröbner bases for public key cryptography. In J. Rafael Sendra and Laureano González-Vega, editors, *ISSAC*, pages 315–324. ACM, 2008.

[17] Joshua Cannell. Obfuscation: Malware's best friend, March 2013.

[18] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-Box Cryptography and an AES Implementation. In *Proceedings of the 9th International Workshop on Selected Areas in Cryptography (SAC 2002)*, volume 2595 of *Lecture Notes in Computer Science*, pages 250–270. Springer, 2002.

[19] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for DRM applications. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management (DRM 2002)*, volume 2696 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002.

[20] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[21] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[22] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation: tools for software protection. *IEEE Trans. Softw. Eng.*, 28:735–746, August 2002.

[23] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Batch fully homomorphic encryption over the integers. IACR Cryptology ePrint Archive, 2013.

[24] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[25] Des. Data encryption standard. In *In FIPS PUB 46, Federal Information Processing Standards Publication*, pages 46–2, 1977.

[26] Matthew T. Dickerson. *The Functional Decomposition of Polynomials*. PhD thesis, Ithaca, NY, USA, 1989.

[27] T. Dierks and E. Rescorla. RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. Technical report, August 2008.

[28] Jintai Ding. A New Variant of the Matsumoto-Imai Cryptosystem through Perturbation. In *Proceedings of the 7th International Workshop on Theory and Practice in Public Key Cryptography (PKC 2004)*, volume 2947 of *Lecture Notes in Computer Science*, pages 305–318. Springer-Verlag, 2004.

[29] Jintai Ding and Jason E. Gower. Inoculating multivariate schemes against differential attacks. In *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2006.

[30] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *Applied Cryptography and Network Security Conference – ACNS 2005*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175, 2005.

[31] Jintai Ding and Bo-Yin Yang. Multivariates polynomials for hashing. In *Information Security and Cryptology Conference – Inscrypt 2007*, pages 358–371. Springer-Verlag, 2008.

[32] Stephen Drape, Clark Thomborson, and Anirban Majumdar. Specifying imperative data obfuscations. In *Information Security*, pages 299–314. Springer, 2007.

[33] Vivien Dubois, Pierre-Alain Fouque, Adi Shamir, and Jacques Stern. Practical cryptanalysis of sflash. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.

[34] Vivien Dubois, Pierre-Alain Fouque, and Jacques Stern. Cryptanalysis of sflash with slightly modified parameters. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 264–275. Springer, 2007.

[35] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139(1–3):61–88, June 1999.

[36] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, ISSAC '02, pages 75–83, New York, NY, USA, 2002. ACM.

[37] J.-C. Faugère and C. Mou. Fast Algorithm for Change of Ordering of Zero-dimensional Gröbner Bases with Sparse Multiplication Matrices. In *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, ISSAC '11, pages 115–122, New York, NY, USA, 2011. ACM.

[38] Jean-Charles Faugère, Patrizia M. Gianni, Daniel Lazard, and Teo Mora. Efficient computation of zero-dimensional gröbner bases by change of ordering. *J. Symb. Comput.*, 16(4):329–344, 1993.

[39] Jean-Charles Faugère and Ludovic Perret. Polynomial equivalence problems: Algorithmic and theoretical aspects. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 30–47. Springer, 2006.

[40] Jean-Charles Faugère and Ludovic Perret. An efficient algorithm for decomposing multivariate polynomials and its applications to cryptography. *J. Symb. Comput.*, 44(12):1676–1689, 2009.

[41] Jean-Charles Faugère and Ludovic Perret. High order derivatives and decomposition of multivariate polynomials. In *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pages 207–214, New York, NY, USA, 2009. ACM.

[42] Jean-Charles Faugère, Joachim von zur Gathen, and Ludovic Perret. Decomposition of generic multivariate polynomials. In *ISSAC '10: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pages 131–137, New York, NY, USA, 2010. ACM.

[43] Michael Fellows and Neal Koblitz. Combinatorial cryptosystems galore! In *Proceedings of the Second International Symposium on Finite Fields, Las Vegas, Nevada, August, 1993*, volume 168 of *Contemporary Mathematics*, pages 51–61. American Mathematical Society, 1994.

[44] Pierre-Alain Fouque, Gilles Macario-Rat, and Jacques Stern. Key recovery on hidden monomial multivariate schemes. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 2008.

[45] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[46] Harvey L. Garner. The residue number system. *Electronic Computers, IRE Transactions on*, EC-8(2):140–147, June 1959.

[47] Craig Gentry. Fully homomorphic encryption using ideal lattices, June 2009.

[48] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Ring switching in bgv-style homomorphic encryption. Cryptography for Networks, SCN'12, 2012.

[49] Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of White Box DES Implementations. In *Proceedings of the 14th International Workshop on Selected Areas in Cryptography (SAC 2007)*, volume 4876 of *Lecture Notes in Computer Science*, pages 278–295. Springer-Verlag, 2007.

[50] David Grant, Kate Krastev, Daniel Lieman, and Igor Shparlinski. A public key cryptosystem based on sparse polynomials. In *Proceedings of an Internatianl Conference*, pages 114–121, New York, April 1998. Springer, Berlin Heidelberg. Available at `http://www.comp.mq.edu.au/~igor/GKLS.ps`.

[51] Dennis Hofheinz and Rainer Steinwandt. A "differential" attack on polly cracker. In *Proceedings of 2002 IEEE International Symposium of Information Theory ISIT*, page 211, 2002. extended abstract.

[52] Hideki Imai and Tsutomu Matsumoto. Algebraic methods for constructing asymmetric cryptosystems. In Jacques Calmet, editor, *AAECC*, volume 229 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 1985.

[53] Matthias Jacob, Dan Boneh, and Edward W. Felten. Attacking an Obfuscated Cipher by Injecting Faults. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management (DRM 2002)*, volume 2696 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2002.

[54] S. Josefsson. Rfc 4648 - the base16, base32, and base64 data encodings, October 2006.

[55] Mohamed Karroumi. Protecting White-Box AES with Dual Ciphers. In *ICISC*, volume 6829 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 2010.

[56] Hamilton E. Link and William D. Neumann. Clarifying Obfuscation: Improving the Security of White-Box DES. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2005)*, volume 1, pages 679–684, Washington, DC, USA, 2005. IEEE Computer Society.

[57] Tsutomu Matsumoto and Hideki Imai. Public quadratic polynomial-tuples for efficient signature-verification and message encryption. In Christoph G. Gunther, editor, *Advances in cryptology — EUROCRYPT '88: Workshop on the Theory and Application of Cryptographic Techniques, Davos, Switzerland, May 25–27, 1988: proceedings*, volume 330, pages 419–453, 1988. Sponsored by the International Association for Cryptologic Research.

[58] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[59] Wil Michiels, Paul Gorissen, and Henk D.L. Hollmann. Cryptanalysis of a Generic Class of White-Box Implementations. In *Proceedings of the 15th International Workshop on Selected Areas in Cryptography (SAC 2008)*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[60] Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao - Lai White-Box AES Implementation. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography, 19th Annual International Workshop, SAC 2012*, volume 7707 of *Lecture Notes in Computer Science*, pages 34–49, Windsor,Ontario,Canada, 2012. Springer-Verlag.

[61] Yoni De Mulder, Peter Roelse, and Bart Preneel. Revisiting the BGE Attack on a White-Box AES Implementation. Cryptology eprint archive, 2013.

[62] Yoni De Mulder, Brecht Wyseur, and Bart Preneel. Cryptanalysis of a Perturbated White-box AES Implementation. In Kishan Chand Gupta and Guang Gong, editors, *Progress in Cryptology - INDOCRYPT 2010*, volume 6498 of *Lecture Notes in Computer Science*, pages 292–310, Hyderabad ,IN, 2010. Springer-Verlag.

[63] Jacques Patarin. Cryptoanalysis of the matsumoto and imai public key scheme of eurocrypt'88. In Don Coppersmith, editor, *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 1995.

[64] Jacques Patarin. Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms. In *Advances in Cryptology - EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 33–48. Springer-Verlag, 1996.

[65] Jacques Patarin and Louis Goubin. Asymmetric cryptography with S-Boxes. In *Proceedings of the First International Conference on Information and Communication Security (ICICS 1997)*, pages 369–380, London, UK, 1997. Springer-Verlag.

[66] Rainer Steinwandt, Willi Geiselmann, and Regine Endsuleit. Attacking a polynomial-based cryptosystem: Polly cracker. *Int. J. Inf. Sec.*, 1(3):143–148, 2002.

[67] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers, June 2010.

[68] Brecht Wyseur. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, 2009.

[69] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In *Proceedings of the 14th International Workshop on Selected Areas in Cryptography (SAC 2007)*, volume 4876 of *Lecture Notes in Computer Science*, pages 264–277. Springer-Verlag, 2007.

[70] Yaying Xiao and Xuejia Lai. A Secure Implementation of White-Box AES. In *Proceedings of the 2nd International Conference on Computer Science and its Applications (ICCSA 2009)*, pages 1–9. IEEE, 2009.

[71] DingFeng Ye, Kwok-Yan Lam, and Zong-Duo Dai. Cryptanalysis of "2R" Schemes. In *Advances in Cryptology - CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 315–325. Springer-Verlag, 1999.

[72] Amr M. Youssef and Guang Gong. Cryptanalysis of imai and matsumoto scheme B asymmetric cryptosystem. In C. Pandu Rangan and Cunsheng Ding, editors, *Progress in Cryptology - INDOCRYPT 2001, Second International Conference on Cryptology in India, Chennai, India, December 16-20, 2001, Proceedings*, volume 2247 of *Lecture Notes in Computer Science*, pages 214–222. Springer, 2001.

[73] William Zhu and Clark Thomborson. A provable scheme for homomorphic obfuscation in software security. In *The IASTED International Conference on Communication, Network and Information Security, CNIS.*, volume Vol. 5., 2005.

[74] William Zhu, Clark D Thomborson, and Fei-Yue Wang. Obfuscate arrays by homomorphic functions. In *GrC*, pages 770–773, 2006.