Advanced Software Protection:
Integration, Research and Exploitation

D2.08
ASPIRE Offline Code Protection Report

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	1st November 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10
Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D2.08/ 1.0
WP and tasks contributing:	WP 2 / Tasks T2.1, T2.2, T2.3, T2.4, T2.5
Due date:	October 2015 - M24
Actual submission date:	17 November 2015
Responsible Organization:	UGent
Editor:	Bjorn De Sutter
Dissemination Level:	Public
Revision:	1.0

Abstract:
This deliverable documents the tool support delivered in the corresponding prototype deliverable D2.07 Offline Code Protection Support, and the research undertaken towards that end in WP2 in year 2 of the project. The code protections documented are data obfuscations, white-box cryptography, client-side code splitting, code obfuscation, and anti-tampering.

Keywords:
Data obfuscation, white-box cryptography, client-server code splitting, code obfuscation, anti-tampering

Editor

Bjorn De Sutter (UGent)

Contributors (ordered according to beneficiary numbers)

Bjorn De Sutter, Bart Coppens (UGent)

Brecht Wyseur, Patrick Hachemane (NAGRA)

Roberto Tiella, Mariano Ceccato (FBK)

Andreas Weber (SFNT)

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

Disclaimer

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 609734.

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Executive Summary

This deliverable reports the year two RTD progress in WP2 on the topic of offline protection techniques. Five major sections report on the progress of the five tasks in WP2.

First, the progress with respect to dynamic forms of data obfuscation of Task T2.1 is reported. In these forms, the constants used as mask or modulus in data obfuscations cannot be recovered by attackers deploying only static attacks. FBK has studied a number of alternative methods to obfuscate the constants, i.e., to generate them dynamically during the program's execution, incl. alias-based opaque predicates, graph enumeration, conjectures, and the 3SAT problem. The latter was initially considered as a potential approach to generate code that is hard to analyse statically, but it was found not to scale well: the run-time overhead becomes too big for significant level of protections. So a novel obfuscation scheme based on the NP-complete n-clique problem is proposed, that builds on a reduction of a 3SAT problem to an n-clique problem to generate hard to analyse code.

Second, the progress with respect to white-box cryptography in Task T2.2 is documented. The focus is on the development of NAGRA's White-Box Tool for ASPIRE, and the implementation of two extensions as foreseen in the DoW: time-limited white-box implementations that provide security only for a limited amount of time but that come with acceptable run-time overhead (unlike the provably secure versions developed earlier); dynamic-key white-box AES, in which a server generates obfuscated keys to be sent to the client rather than embedding a fixed key into the client.

Third, SFNT reports what additional background they have contributed to the project for Task T2.3 on the subject of client-server code splitting, thus showing that the goals of that task have been reached.

Fourth, the progress regarding native code obfuscation in Task T2.4 is discussed. UGent reports improvements in the way opaque predicates, branch functions and control flow flattening are applied based on profile information instead of purely stochastically, which results in significantly reduced execution times. Furthermore, they report extensive debugging and intermediate representation bookkeeping functionality to make the binary code obfuscations composible with other protections deployed in the binary-level part of the ACTC. UGent also reports on a new approach to obfuscate control flow by means of externally defined data structures and APIs that allow more stealthy protection. GTO reports the architecture of an algorithm-specific multithreaded AES-based cryptographic technique that can be used to hide master keys embedded in client applications behind complex multithreaded computations. This performance of this technique is also evaluated.

Finally, four techniques are discussed from Task T2.5 in the domain of anti-tampering. UGent presents the first results and implementation effort of a so-called self-debugging anti-debugging technique, in which an application is split over a debuggee and a debugger process to prevent the attachment of an attacker debugger. UGent also reports the initial results obtained for simple call-back checks that can prevent call-backs from attacker-injected code. GTO briefly discusses the first results obtained for control-flow tagging, a code execution integrity verification check, and finally, UGent briefly discusses the first results obtained with respect to offline code guards, which reuses the guards also used in remote attestation (WP3, Task 3.2).

Contents

Section 1 Introduction ... 1
Section 2 Task T2.1: Data Obfuscation ... 2
2.1 Introduction ... 2
2.1.1 Data Obfuscations ... 2
2.1.1.1 XOR Masking ... 2
2.1.1.2 RNC Encoding ... 2
2.2 Threat Model ... 2
2.3 Possible Extensions ... 3
2.3.1 Convert Static to Procedural Data ... 3
2.3.2 Opaque Constant based on 3SAT Problem ... 3
2.3.3 Opaque Constant based on Collberg's Opaque Predicates ... 4
2.3.4 Graphs Enumeration ... 4
2.3.5 Leveraging Conjectures ... 4
2.4 Our Novel Obfuscation Scheme ... 5
2.4.1 Obfuscation Scheme Requirements ... 5
2.4.1.1 Analysis of the 3SAT Approach ... 6
2.4.2 Data Obfuscation as a k-clique Problem ... 8
2.4.3 Reducing 3-SAT to k-clique ... 9
2.4.3.1 Example ... 10
2.4.4 Coding the graph ... 10
2.5 Attack Analysis ... 11
2.5.1 Running the n-clique checking code with KLEE ... 11
2.5.1.1 A Sanity Check on a satisfiable problem ... 12
2.5.1.2 Running KLEE on a unsatisfiable problem ... 13
2.6 Dynamic XOR Masking ... 13
2.6.1 Handling variable initializations ... 14
2.7 Implementation ... 14
2.7.1 Updated Obfuscation Process ... 14
2.7.2 Constant-generating functions creation process ... 15
2.7.2.1 3SAT Problem Generation ... 15
2.7.2.2 3SAT Problem Satisfiability ... 16
2.7.2.3 Graph Generation ... 16
2.7.2.4 Constant-generating Function Forging ... 17
Section 3 Task T2.2: White Box Cryptography ... 18
3.1 Introduction ... 18
3.2 White Box Cryptography ... 19
3.3 White Box Tool for ASPIRE ... 20
3.3.1 Overview ... 21

3.3.2 Technical choices 21
3.3.3 Previous steps 21
3.3.4 M24 achievements 22
3.3.4.1 Overview 22
3.3.4.2 Input data and annotations 23
3.3.4.3 Output data and integration to ACTC 23
3.3.4.4 Examples 23
Section 4 Task T2.3: Client-Side Code Splitting 25
4.1 Automated detection of code regions to split off 25
4.2 X-Translator: 25
4.2.1 Supporting a larger subset of the ARM instruction set 26
4.2.2 Embedding the continuation address 28
4.2.3 Supporting shared objects 29
4.2.4 Post-Linker interface 29
4.2.5 Chunk internal control flow 30
4.2.6 Chunk test framework 30
4.2.7 Integrating the LLVM Interpreter (lli) 31
4.2.8 Supporting memory access 31
4.2.9 Tool versioning and automated release builds 33
Section 5 Task T2.4: Binary Code Obfuscation 35
5.1 Control Flow Obfuscation 35
5.1.1 Improving the deployment of existing obfuscations 35
5.1.2 Flexible, two-way opaque predicates 36
5.2 Multithreaded Cryptography 39
5.2.1 Original AES 40
5.2.2 Master key 40
5.2.3 Architecture 40
5.2.3.1 Crypto server 41
5.2.4 CryptoMultiThread library 42
5.2.5 Application Server 43
5.2.6 Application Performance degradation 43
5.2.7 Limitations 44
Section 6 Task T2.5: Anti-Tampering 45
6.1 Anti-Debugging 45
6.2 Anti-callback Stack Checks 46
6.3 Control Flow Tagging 47
6.4 Code Guards 48
Section 7 List of Abbreviations 50
Bibliography 51

List of Figures

Figure 1 – Using Collatz's conjecture to obfuscate a predicate 5
Figure 2 – 3SAT checking code instrumented to be run using KLEE 6
Figure 3 – KLEE execution time for checking 3SAT problems 8
Figure 4 – A graph derived by reducing a 3SAT problem with 3 variables and 4 clauses. 10
Figure 5 – A 4-clique found by the symbolic executor KLEE 13
Figure 6 – Process to create code for constant-generating functions 15
Figure 7 – Structure of the constant generation function 16
Figure 8 – Dynamic-key white-box high-level view 20
Figure 9 – T2.2 process overview 21
Figure 10 – T2.2 process overview - dynamic key 22
Figure 11 – Dynamic key protection 23
Figure 12 – Overhead comparison between stochastic and profile-guided obfuscation 35
Figure 13 – Principle of flexible two-way opaque predicates 36
Figure 14 – Program size overhead of using flexible two-way opaque predicates. 38
Figure 15 – Execution times for flexible two-way opaque predicates 39
Figure 16 – Architecture of the Multi-Threaded Crypto protection 41
Figure 17 – AES 42
Figure 18 – Modified AES 43
Figure 19 – Permutation rule 43
Figure 20 – Processing time with 128bit key for plain text sizes for 1, 3, 4 and 7 threads 44
Figure 21 – Comparison of the two methods to transform memory accesses. 46

List of Tables

Table 1 – KLEE execution times. 7
Table 2 – XOR Masking parameter generation function 14
Table 3 – Static vs Dynamic XOR Masking...14
Table 4 – Overview of the benchmarks use to evaluate flexible opaque predicates.............38

Section 1 Introduction

This deliverable reports the progress made in the five tasks of WP2 Offline Software Protections. Sections 2 to 6 are devoted to tasks T2.1 to T2.5.

Tasks T2.1 to T2.4 are supposed to end in M24 according to the DoW. This means that the basic research into the different techniques, and the (isolated) tool support development need to be finished. As foreseen in the DoW, a minimal amount of activity is still to be expected in year 3, however, as the techniques are further integrated into the ASPIRE Compiler Tool Chain, as they are deployed on the project use cases, and as they will be driven by the ASPIRE Decision Support System in year 3.

This deliverable D2.08 of type report also documents the tool support that has been developed in year 2 of the project in WP2, and that is delivered as prototype deliverable D2.07.

Section 2 Task T2.1: Data Obfuscation

Section Authors: Roberto Tiella, Mariano Ceccato

2.1 Introduction

At the end of the first year of the ASPIRE project, the output of task T2.1 (described in deliverable D2.01 "Early White-Box Cryptography and Data Obfuscation Report") consisted of the implementation of four algorithms for data obfuscation taken from the state of the art. They are:

- XOR Masking;
- Variable Merging;
- Residue Number Coding (RNC); and
- Convert Static to Procedural Data.

The objective of the second year of the project is to extend (a subset of) these algorithms towards stronger variants.

2.1.1 Data Obfuscations

In this subsection, we recall some concepts from D2.01 for the reader's sake. The reader is suggested to refer to D2.01 in case further details are needed.

A *data obfuscation transformation* is a program transformation aimed at hiding variables' values. A data obfuscation transformation is characterised by a function $e(v)$ (called *encoding function* in what follows) that describes how the transformation acts on the values assumed by obfuscated variables. In the following two subsections we present two examples of data obfuscation, namely XOR Masking and RNC Encoding.

2.1.1.1 XOR Masking

A XOR Masking transformation is characterised by the following encoding function:

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

Where $\oplus$ is the bit-wise XOR operator and p is a fixed parameter called *the mask*.

2.1.1.2 RNC Encoding

Given a set of n integers $\{m_1, m_2, \ldots, m_n\}$ pairwise mutually prime (two numbers are *mutually prime* if their only common divisor is 1), RNC encoding function is defined as:

$$e(v) = (v \bmod m_1, v \bmod m_2, \ldots, v \bmod m_n)$$

The function $e(v)$ is guaranteed invertible under the assumptions and the original value v can be decoded back from an n-tuple $(y_1, y_2, \ldots, y_n)$ by means of the extended Euclid greatest common divisor algorithm. Integers $\{m_1, m_2, \ldots, m_n\}$ are called *the modules* in what follows.

2.2 Threat Model

The motivation to elaborate an extension to the previous work is that state-of-the-art data obfuscations are vulnerable to attacks based on static analysis. In fact, masks used in XOR Masking and modules used in Residue Number Coding, for example, are static constants

that, once identified in the static code, can be used to decode (i.e., obtain the clear value) of obfuscated variables.

A way to turn these obfuscation schemes more reliable is to make it harder to detect these constants by static analysis.

Dynamic analysis would be probably always an issue for data obfuscation, because the clear values can be intercepted at run time after decoding. Other protections are required to avoid run-time monitoring, such as anti-debugging.

We assume the subsequent attack model:

- The attacker adopts static analysis, i.e., they have full access to the compiled code and can run state-of-the-art analysis tools and algorithm on it;
- The attacker can run the code (or part of it), but they cannot perform dynamic analysis on it, e.g., a debugger cannot be attached to the code.

2.3 Possible Extensions

This section collects the results of discussions and of brainstorming sessions, towards the definition of the novel extension.

2.3.1 Convert Static to Procedural Data

The first extension consists of removing the plain constant used in the data obfuscation (mask or module) and replacing it with a procedure to compute it on demand at run time, for example by resorting to a Mealy Machine as described by C. Collberg and J. Nagra [Col09]. In this way, a simple search in the code binaries would not succeed. However, the procedure should not be trivial, otherwise an attacker could figure out its behaviour and forecast the result.

Moreover, advanced static analysis can break non-trivial obfuscations, when the analysis is able to statically figure out the outcome of non-trivial procedures to compute the constant. An example of such powerful static analysis is symbolic execution. Therefore, the obfuscation should be designed to be strong against advanced static analyses, for example by requiring the analysis to solve intractable problems to break the obfuscation.

The obfuscation should be designed such that, to break it, a static analysis tool should solve a problem known to be intractable.

Moreover, if the attacker figures out that the output of this procedure does not depend on the program input (or on random values computed at run time), the attacker could run this procedure once and then reuse the result to break the obfuscation.

The procedure to compute the constant should depend on program input (including random values computed at run time).

2.3.2 Opaque Constant based on 3SAT Problem

The work by Moser et al. [Mos07] describes how to turn static constants into *opaque constants*, i.e. constant values that are difficult to guess statically. In their approach, to detect the value of the constant, an attacker would need to solve the satisfiability problem for a Boolean formula in 3 variables (3SAT), a problem that is known to be NP-complete.

They use opaque constants (i) to hide absolute and relative jumps/calls to make the control flow graph very hard to recover; (ii) to hide the address of program variables, and in particular the import table in dynamic library headers; and (iii) hide variable usage by breaking def-use chains.

Moser et al. [Mos07] used this obfuscation to make known malware stealth to commercial malware detectors and to advanced semantic-based malware detectors that still resort to

static analysis. They claim for the need of more advanced malware detection based on dynamic analysis.

In the paper, they describe their algorithm to generate an opaque bit. We could extend this approach to generate a sequence of 32 opaque bits that would encode integer values used as masks or modules in obfuscations. Indeed, different 3SAT formulas could be used to generate the same constant, thus complicating the attacker analysis.

2.3.3 Opaque Constant based on Collberg's Opaque Predicates

Collberg et al. [Col98] propose a technique to forge opaque predicates. The technique leverages the hardness (undecidability, in general [Ram94]) of the statically must/may point-to analysis problem. The very basic idea is to have two pointers running on two disjointed components of a dynamic data structure such as a graph. Involved dynamic data structures are updated in certain point of program execution, randomly adding new components or removing existing components. An articulated example that leverages the idea is sketched in the article. The article recommends developers to provide many implementation variants, obfuscated, and merged with actual code.

Leveraging the proposed technique, each bit of the opaque constant is decided based on an opaque predicate.

2.3.4 Graphs Enumeration

Collberg et al. [Col99] describe watermarking techniques. In their paper, the authors suggest to encode constants into data structures by means of *enumeration*. Graph-based structures such as trees and circular lists can be systematically enumerated, actually establishing a link from an integer number to the "shape" of an instance of a specific data structure. Algorithms to support this protection are presented in Yong He's Master Thesis [He02], the work by Palsberg et al. [Pal00] and more recently by Chron and Nikolopoulos [Chr11].

2.3.5 Leveraging Conjectures

Wang et al. [Wan11] present a technique to obfuscate predicates that trigger malware behaviours. The technique aims at preventing to recover which conditions make a predicate true by means of an analysis based on symbolic execution. Conditions are defined in terms of values assumed by some input variables. They leverage some mathematical conjectures, for example the Collatz's one. Collatz's conjecture says that the sequence $\{y_k\}$ defined by:

$$y_0 \in N; y_0 > 0$$

$$y_{k+1} = \begin{cases} 3y_k + 1 & if \quad y_k \equiv 1 \pmod 2 \\ \frac{y_k}{2} & if \quad y_k \equiv 0 \pmod 2 \end{cases}$$

eventually reaches 1. The conjecture was proven true for $y_0 \leq \sim 2^{58}$ by computation but no formal proof is available of its validity.

Figure 1 shows an example based on the Collatz's conjecture. In the program on the left, x=30 triggers the condition x==30 and causes the malware to be executed. In the program in the centre of the figure, supposing the conjecture true, the loop is eventually exited. The figure on the right depicts how to embed the trigger into the loop: when y reaches value 1 the condition is true and the malware is executed.

```
// x given as input

if (x == 30) {
  some_malware();
}
```

```
y = // any integer

while (y>1) {
  if (y % 2 == 1) {
    y = 3*y+1;
  } else {
    y = y/2;
  }
}
```

```
// x given as input

int y = x + 1000;

while (y>1) {
  if (y % 2 == 1){
    y = 3*y+1;
  } else {
    y = y/2;
  }

  if (x-y>28 && x+y<32){
    some_malware();
    break;
  }
}
```

Figure 1 – Using Collatz's conjecture to obfuscate a predicate

2.4 Our Novel Obfuscation Scheme

This section illustrates the new obfuscation scheme we devised during the survey of the existing literature.

2.4.1 Obfuscation Scheme Requirements

The discussion of data obfuscation based on the 3SAT problem [Mos07] highlighted what are the requirements for the mathematical problem to be used for our new obfuscation scheme:

1. **Difficult for the attacker to analyse**: Undoing obfuscation (i.e., recovering the obfuscated constant) should require the attacker to solve a problem Pr, known to be NP-complete that requires a non-polynomial time.
2. **Opaqueness of the problem**: The solution v is build staring from random/input values. It means that, the obfuscation transformation consists of generating an instance of the problem Pr, whose solution v depends on and can be built starting from any input/random data.
3. **Easy for the defender (at run time)**: The obfuscated program can compute easily (in polynomial time) the obfuscated value. It particular, it is fast to check that a value v is the solution of the problem Pr.

The 3SAT problem satisfies these requirements because it is:

1. **Difficult for the attacker to analyse**: Given a formula, it is difficult to understand if it is satisfiable (unsatisfiable), and what are the variable values that make it TRUE (FALSE);
2. **Opaqueness of the problem**: It is easy to construct a (hard to solve) formula that always evaluate to TRUE or to FALSE. In this way, the variables to use in the formula can trivially depend on random/input values.

```
int v1,nv1;
...

int * l1[17] = { &nv2, ...};
int * l2[17] = { &nv3, ... };
int * l3[17] = { &nv4, ... };

void init_klee() {
        klee_make_symbolic(&v1,sizeof(int),"v1");
        klee_assume(v1 == 0 | v1 == 1);
        nv1=1-v1;
        ...
}

void main(...) {
        int res = 1;
        int i;

        init_klee();

        for (i=0; i<NC; i++) {
        if (!*l1[i] && !*l2[i] && !*l3[i]) {
                res = 0;
                 break;
        }
        printf ("truth value=%d\n",sat);
}
```

Figure 2 – 3SAT checking code instrumented to be run using KLEE

3. **Easy for the defender (at run time)**: Given concrete Boolean values of formula variables, the formula evaluation is fast.

2.4.1.1 Analysis of the 3SAT Approach

As stated by Moser et al. [Mos07], a static analyser that aims to determine exactly the possible values of an opaque constant has to solve an instance of the 3SAT problem. We evaluated the approach proposed by Moser et al. [Mos07] using the symbolic executor KLEE [Cad08]. KLEE is a symbolic virtual machine built on top of the LLVM [Lat04] compiler infrastructure. KLEE is able to run a C program symbolically provided the source code is modified to declare variables that have to hold symbolic values. KLEE can be used to recover the opaque constant, because KLEE will try to identify the set of input that solve the 3SAT problem and leak the value of the opaque constant.

We run KLEE on the code presented by Moser et al. in their paper [Mos07]. For the reader's sake we sketched the code in Figure 2. The original code checks whether a 3SAT formula encoded in vectors l1,l2 and l3, is true under the assignment of some random values to variables v1,v2,...,vn (only declaration for v1 is shown in the figure. If the formula is true the

program will print “truth value=1”, “truth value=0” is printed otherwise. The figure shows the code modified to accommodate KLEE’s declarations of symbolic variables.

As KLEE attempts to cover all paths in the application it tries to find concrete replacements for symbolic values that causes the program both to print “truth value=1” and “truth value=0”. This is equivalent to solve a 3SAT problem.

Having fixed the number NVARS of Boolean variables in a 3SAT problem, not all problems are equally difficult in terms of required time for finding a solution. Selman et al. [Sel96] have shown that if we randomly draw a SAT3 formulas with NCLS clauses and test for its satisfiability, setting NCLS to floor(4.3*NVARS) gives a high probability of choosing a difficult 3SAT problem. We adopted this finding in this analysis.

Table 1 reports the user time (UTIME, in seconds, and its standard deviation SD) required by KLEE to run on unsatisfiable 3SAT problems with a number of variables NVARS ranging from 4 to 20 and floor(4.3*NVARS) clauses. It corresponds to the time required by static analysis to break data obfuscation based on 3SAT, as proposed by Moser at al. The boxplot in Figure 3 shows time needed to break obfuscation (UTIME) whit an increasing number variables in the 3SAT formula (NVARS).

Table 1 – KLEE execution times.

NVARS	Runs	UTIME	SD
4	10	0.33	0.02
8	10	1.54	0.17
12	10	8.68	1.63
16	10	56.77	22.9
20	10	513.56	172.03

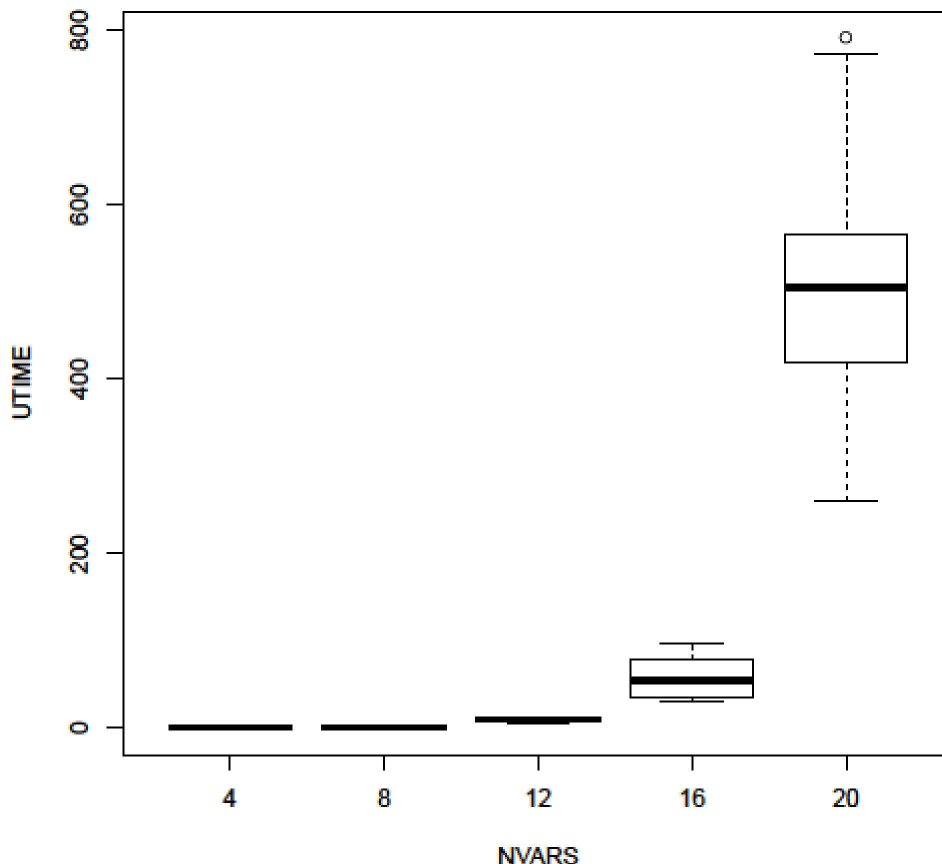

Figure 3 – KLEE execution time for checking 3SAT problems

As we can see, even if the 3SAT problem is NP-complete, it can be practically solved in a limited amount of time using available static analysis tools (in our case KLEE). A problem with 16 variables can be solved in less than one minute, and a problem with 20 variables requires less than 7 minutes.

In the graph shows an evident exponential trend of the time required to break the 3SAT obfuscation, so one might think of using an arbitrarily big 3SAT problem to make the attack time diverge. However, to keep the obfuscation overhead manageable, in their implementation Moser at al. adopted a rather small problem size. In their empirical assessment, they considered a 3SAT problem with 20 clauses. It recorded an increased program size of 30 times and an execution time of almost five times longer (+471%). However, 20 clauses means approximately 5 variables, a size that we could break in less than one second.

In the following, we will present our approach to a novel data obfuscation scheme that, as the 3SAT, is still based on a NP-complete problem in order to satisfy the three obfuscation requirements (difficult for the attacker, opaqueness of the problem and easy for the defender). However, with the complexity comparable to 3SAT, our approach is meant to be more robust against static analysis. In particular, our approach is based on the k-clique problem.

2.4.2 Data Obfuscation as a k-clique Problem

Karp [Kar96] lists 21 NP-complete problems and Garey and Johnson's book [Gar79] contains tens of NP-complete examples from graph theory, sets and partitions, sequencing and

scheduling, just to mention some examples. The NP-complete problem we decided to use is the k-clique problem:

k-clique problem: *given a graph G, does it contain a clique of size k?*

A clique of size k is defined as a *complete* subgraph G' of G of size k and a graph G is said to be *complete* if every two distinct vertexes in G are connected.

The k-clique problem is included in Karp's collection of NP-complete problems and, in the same work, the problem is proven to be NP-complete by showing a straightforward reduction from SAT. We propose to use the same reduction from SAT to k-clique, described below, to construct a k-clique problem starting from an arbitrary SAT problem. The satisfiability of a k-clique problem constructed in this way depends on the satisfiability of the starting SAT problem.

The proposed obfuscation scheme leverages the k-clique problem intractability in this way:

- We generate a 3SAT unsatisfiable formula f in conjunctive normal form, reusing the results by Selman et al. [Sel96]
- We construct a k-clique problem by reduction from the 3SAT problem: a solution to the k-clique problem is a solution to the satisfiability problem on f, so we know that it is NP-complete;
- We take a random subset S of k nodes of G. It is fast to compute whether S is a clique. This verification takes k(k−1)/2 checks in the worst case, namely when S is actually a clique, but on the average case, when S is not a clique, it takes less, because the check can stop when the first missing edge is found;
- By construction, we know that G does not contain cliques of size k (otherwise f would be satisfiable). So we know that s is not a clique and the check will return false.

This approach is used to generate one opaque bit. Therefore, for a 32-bit constant we need 32 k-clique problems generated according to the previous algorithm.

This opaque constant can be used to:

- Hide a cryptographic key;
- Generate the module of RNC or the mask for XOR once at program initialization time and then keep it in memory; or
- Generate the module of RNC or the mask for XOR every time it is required, and then discard and overwrite the value.

2.4.3 Reducing 3-SAT to k-clique

Following the original Karp reduction scheme [Kar96], given the 3-SAT formula in *m* variables $v_1, v_2, \ldots, v_m$ consisting in *n* clauses:

$$\phi = \bigwedge_{i=1,\ldots,n} \alpha_{i,1} \vee \alpha_{i,2} \vee \alpha_{i,3}$$

with:

$$\alpha_{i,j} = \begin{cases} v_k & or \\ \neg v_k \end{cases}$$

it is possible to construct the following graph $G_\Phi = (V, E)$, where:

$$V = \{(i,\alpha_{i,1}),(i,\alpha_{i,2}),(i,\alpha_{i,3}) \mid i=1,\ldots,n\}$$

and

$$((i_1,\alpha_{i_1 j_1}),(i_2,\alpha_{i_2 j_2}))\in E \text{ iff } i_1 \neq i_2 \text{ and } \alpha_{i_1 j_1} \wedge \alpha_{i_2 j_2} \text{ satisfiable.}$$

It follows by construction that the 3-SAT formula Φ is satisfiable if and only if the graph G_Φ has a *n*-clique (*n* the number of clauses in the formula). A 3SAT formula Φ in *m* variables and *n* clauses is mapped in a graph which has 3*n* nodes and a number of arcs which is bounded by $9n^2$.

2.4.3.1 Example

The following (satisfiable) logical formula:

$$(\neg v_1 \vee \neg v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_2 \vee \neg v_3) \wedge (v_1 \vee \neg v_2 \vee v_3) \wedge (v_1 \vee v_2 \vee v_3)$$

maps to the graph depicted in Figure 4.

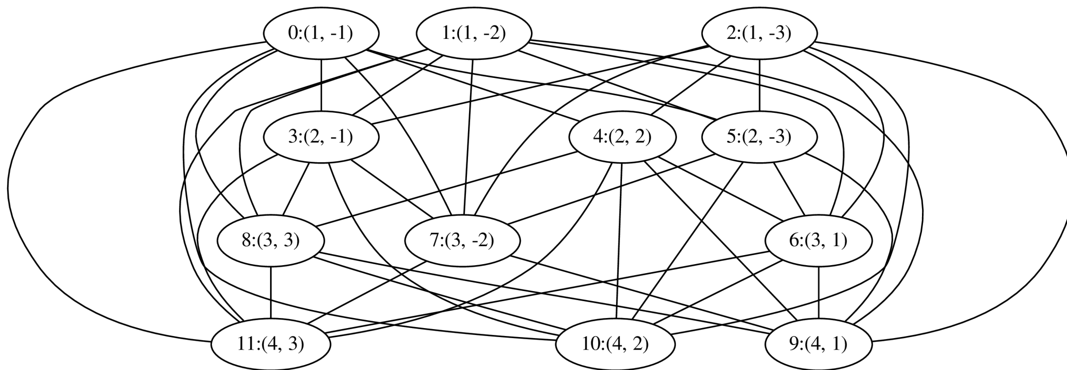

Figure 4 – A graph derived by reducing a 3SAT problem with 3 variables and 4 clauses.

In the figure, each vertex is labelled following the syntax “<id>:(c, l)” where id is a vertex identifier, c is the clause index and l is a positive integer k if the literal v_k is present in the clause c or –k if the literal v_k is present negated in the clause c.

It can be easily checked from the figure that, among many other solutions, the set of nodes {0,3,7,11} forms a 4-clique in the given graph. The clique defines the following assignment:

$$v_1 = False$$
$$v_2 = False$$
$$v_3 = True$$

It is easy to verify that the above assignments make the formula true.

2.4.4 Coding the graph

The graph is coded in the C programming language by means of an adjacency matrix:

```
int m[][];
```

where `m[i][j] == 1` means i-th node and j-th node are connected by an edge while the edges are not connected if `m[i][j] == 0`. The following piece of code checks if a random subset of *n* nodes forms an *n*-clique:

```
1: int res = 0;
2: int i,j,k;

3: int * idx = malloc(n*sizeof(int));

4: assign_randomly(n,idx);

5: for (i=0; i<n-1; i++)
6:  for (j=i+1; j<n-1; j++)
7:     if (!m[idx[i]][idx[j]])) {
8:          res = 1;
9:          break;
10:     }

11: free(idx);

12: if (!res) {
13:   // this branch is taken if idx identifies a n-clique
14: }
```

where `assign_randomly(n,v)` is a function which assigns to the vector `v` `n` unique random values in the range [0,...,$n-1$], i.e. for all i, $0 \le v(i) < n-1$ and for all $i \neq j$, $v(i) \neq v(j)$. In our settings, `assign_randomly(m,n,v)` is implemented by means of the Knuth shuffle [Knu69].

If the 3-SAT formula is unsatisfiable, we know that the true branch of the if-statement at line #12 will never be executed. Trying to run such code symbolically, a symbolic executor will be trapped in solving a NP-complete problem trying to find a way to traverse the unfeasible branch.

2.5 Attack Analysis

In this section we present how we tested our approach using KLEE using the same methodology employed to analyse the 3SAT approach in Section 2.4.1.1.

2.5.1 Running the n-clique checking code with KLEE

We simulated the task of an attacker running KLEE against the *k*-clique checking. Actually this task is by far simpler than the one an actual attacker has to perform as in the real case, for example, a symbolic executor on binary code has to be executed or the binary has to be decompiled to C language before attempting the analysis.

We replaced the random generation procedure listed above between lines #15 and #33 with the following code, containing a KLEE declaration function (`klee_make_symbolic`) and constraints (`klee_assume`). Furthermore dynamic allocation of memory is replaced by static allocation:

```
15': klee_make_symbolic(idx,NODE_NUM*sizeof(int),"idx");

16': for (i=0; i<NODE_NUM; i++) {
17':   klee_assume((idx[i] >= 0) & (idx[i] < NODE_NUM));
18': }

19': for (i=0; i<NODE_NUM; i++) {
```

```
20':    for (j=0; j<NODE_NUM; j++) {
21':      if (i != j) {
22':        klee_assume(idx[i] != idx[j]);
23':      }
24':    }
25': }
```

Line #15' declares idx as a symbolic array of NODE_NUM ints. For-loops from line #16' to line #25' are used to state that idx is a permutation of the sequence {0,1,2,…,NODE_NUM-1}.

2.5.1.1 A Sanity Check on a satisfiable problem

Before performing the actual experiment, we need to check that KLEE is the right tool to address the problem of breaking our obfuscation. To achieve this objective, we try and use KLEE to solve a very small k-clique problem that is satisfiable, i.e., for which a clique of size k exists. It should be noted, however, that the problem used in the sanity check is (i) smaller than the problem that we will use to obfuscate data and (ii) our obfuscation scheme is based on a probably-hard to verify un-satisfiable formula (as described by Selman et al. [Sel96]), much harder to address on average than a randomly generated satisfiable formula, because in common solver implementations the latter requires a more exhaustive search.

As a sanity check we run KLEE on various k-clique problems derived from satisfiable 3SAT formulas. For example, on the 4-clique problem defined after the (satisfiable) 3SAT formula shown above `KLEE` produces two test cases, the second one consisting of:

```
ktest file : 'klee-last/test000002.ktest'
args       : ['main.bc']
num objects: 2
object    0: name: 'model_version'
object    0: size: 4
object    0: data: 1
object    1: name: 'idx'
object    1: size: 48
object    1: data: '\x00\x00\x00\x00\x03\x00\x00\x00\x0b\x00\x00\x00\x07\
x00\x00\x00\n\x00\x00\x00\x01\x00\x00\x00\t\x00\x00\x00\x02\x00\x00\x00\x06
\x00\x00\x00\x05\x00\x00\x00\x04\x00\x00\x00\x08\x00\x00\x00'
```

From lines starting with 'object 1:' we can recover the values of the vector idx:

0,3,11,7,10,1,9,2,6,5,4,8

The first four elements of 'object 1' identify a 4-clique in the graph. Figure 5 shows the identified subgraph.

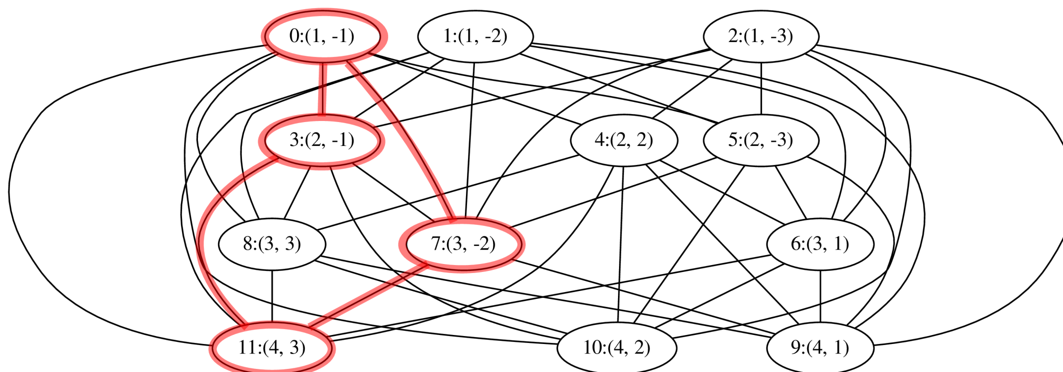

Figure 5 – A 4-clique found by the symbolic executor KLEE

The result of the sanity check allows us to consider KLEE as the right tool to break data obfuscation, it applies static analysis to elaborate a solution to the k-clique problem.

2.5.1.2 Running KLEE on a unsatisfiable problem

In the actual experimental assessment, we run the KLEE symbolic executor on an k-clque (unsatisfiable) problem with 4 variables and 17 clauses. However, the problem is so complex to solve that KLEE is unable to conclude the analysis on a Linux 64-bit machine, 8 cores Intel i7, with 6 GBytes of RAM. Executions ended with an out of memory error after approximately 8 hours.

Our intention was to plot the analysis time increase for problems with different the number of variables (as we did for SAT). However, the obfuscation is so hard to break that a state-of-the-art static analysis tools such KLEE, fails even on the smallest problem size.

Our conjecture is that our mapping from SAT to k-clique is very hard to revert. Our intuition is that KLEE applied to the k-clique problem is not able to recover the original SAT problem, but a much more complex one (usually solvers always work with SAT formulas), and this requires too much time to be solved.

Eventually, our novel data obfuscation scheme overcomes the weakness problem that we detected on 3SAT by Moser at al. (static analysis could, in fact, break it) but sill satisfy by construction the three obfuscation requirements.

In fact, while KLEE was effective in breaking opaque constants based on 3SAT, the same tool could not break opaque constants based on k-clique, even at the smallest problem size.

2.6 Dynamic XOR Masking

In this section we present the second major improvement we performed to data obfuscation techniques developed during the first year of the ASPIRE project, namely the “Dynamic XOR Masking” technique.

As remembered in the introduction, a XOR Masking transformation is defined as:

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

Dynamic XOR Masking is a variant of XOR Masking, where the mask *p* which is involved in the transformation is defined at run time instead of being statically decided at obfuscation time. Thus, the encoding function becomes:

$$e(v) = v \oplus dynmsk_v()$$

where *dynmsk_v()* is a function that returns a randomly drawn number at the first invocation and keeps returning the same number on successive invocations. As an example, Dynamic XOR Masking can be used to protect code from multiple memory scans across executions

because masks are randomly changed at every new run. The code for the function *dynmsk_v()* follows:

Table 2 – XOR Masking parameter generation function

```
static TYPE m = -1;
TYPE dynmsk_v() {
      if (m == -1) {
            m = random() % MAXTYPE;
      }
      return m;
}
```

Depending on the size of the obfuscation variable *v*, namely char, short, integer, or long, constants TYPE and MAXTYPE are defined accordingly. Table 3 shows (a) a simple C code snippet (on the left), (b) a version obtained applying static XOR Masking (in centre) and (c) a version obtained applying dynamic XOR Masking (on the right). The clear code consists in two variable initializations and one statement involving an addition. In the static variant, we decided to use a masks 10, 11 and 12 respectively to encode x, y and z values. In the dynamic case those constant masks are replaced with calls to functions dynmsk_x, dynmak_y and dynmsk_z respectively.

Table 3 – Static vs Dynamic XOR Masking

Clear Code	Static XOR masking	Dynamic XOR Masking

`1: x = 3;`
`2: y = 8;`
`3: z = x + y;`
	`x = 3^10;`
	`y = 8^11;`
	`z = ((x^10)+(y^11))^12;`
		`x = opcnst_3()^dynmsk_x();`
		`y = opcnst_8()^dynmsk_y();`
		`z = ((x^dynmsk_x())+`
		`y^dynmsk_y()))^dynmsk _z();`

2.6.1 Handling variable initializations

Often a variable is initialized with some constant, like in lines 1 and 2 of the example in Table 3. In applying XOR Masking, such constants are replaced with XOR expressions. In the static case, variable initialization XOR expressions such as “3^10” are evaluated by the compiler at compiling time and original constants, namely 3 in the example, are no more present in the compiled code. This is not the case when dynamic XOR masking is applied. In this case, every constant must be replaced with a call to the opaque constant generating function, to avoid the possibility of recovering constant values by means of inspecting the compiled code.

2.7 Implementation

We implemented the algorithms presented above in a component of the ASPIRE Compile Tool Chain (ACTC), named “Data Obfuscator”. A detailed description of how the Data Obfuscator component was developed is given in D2.01. In the present Section, we describe how we extended such process to include the dynamic variants.

2.7.1 Updated Obfuscation Process

The data obfuscation process is updated according to in the following steps:

1. Variable definitions and uses are obfuscated according to code annotations. Protection requirements are expressed in the source code by using annotations as described in Section 4 of D02.1 “Early White-Box Cryptography and Data Obfuscation Report”;

a. If Dynamic XOR Masking is used, XOR Masking parameters are replaced with calls to *dynmask* functions and constants used in variable initialization with calls to functions (opaque-constant generator function in what follows) that compute the required opaque constants on-the-fly;
b. otherwise constants, used as masks and modulus in encoding and decoding expressions, are replaced with opaque-constant generator functions that compute the required value at run-time;

2. Files containing definitions for the opaque-constant generator functions are created and have to be added to the compilation process.

Step 2 of the process is the heart of the novel obfuscation technique and we present it in more detail next.

2.7.2 Constant-generating functions creation process

Figure 6 depicts the process for generating a function that computes an opaque constant. The input consists of the constant which is supposed to fit an NBITS integer. The output is a file containing the definition of a function which returns the value of the input constant. The value of the opaque constant is the result of the computation described in previous parts of this section. In Figure 6, the black dot represents the beginning of the process. In the rest of this subsection, we will present the whole process.

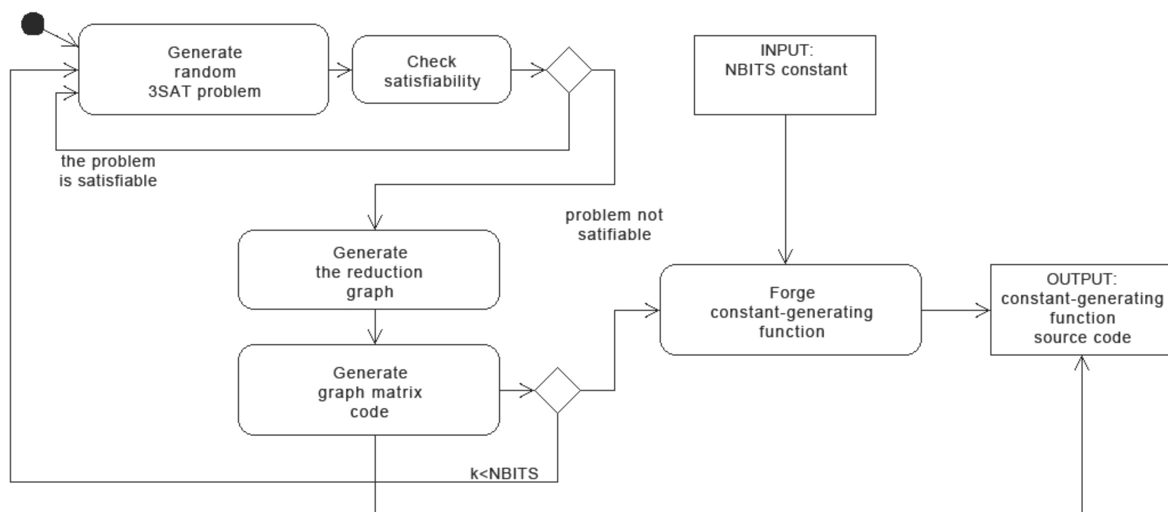

Figure 6 – Process to create code for constant-generating functions

2.7.2.1 3SAT Problem Generation

```
int t_00105_ASPIRE_opaque_constant_13(int n) {
...
int res = 0;
label_0:
  for (k=0; k<n; k++) {
       idx[k] = rand() % s;
  }
  for (i=0; i<n-1; i++) {
    for (j=i+1; j<n-1; j++) {
      if (!t_00105_ASPIRE_opaque_constant_13_m_0[idx[i]][idx[j]]) {
          res += 0*(1<<0); // bit 0 of the constant
          goto label_1;
      }
     }
   }
label_1:
...
return res;
}
```

Figure 7 – Structure of the constant generation function

The first step is the generation of a random initial 3SAT problem. Following the work of Selman et at. [Sel96], we used a generator based on the "fixed clause length model" which is characterised by three parameters: the number N of variables, the number L of literals per clause and the number M of clauses. We left N to be specified by the developer by means of a specific annotation parameter as documented in D5.02 "ASPIRE Offline Compiler Tool Chain". By changing the number of variables N the developer has a mean to tune the hardness of the generated 3SAT problem (and consequently the effort required to tamper with the obfuscated code) versus the amount of memory overhead required to store the related k-clique problem. In our case, L, the number of literals per clause is fixed to the value 3 by definition of 3SAT. The number M of clauses, is fixed to floor(4.3*N) that will produce a hard-to-solve SAT problem with a high probability, as reported by Selman et at [Sel96].

2.7.2.2 3SAT Problem Satisfiability

This SAT problem is then checked using a SAT procedure to verify it is unsatisfiable. The generation step is repeated until an unsatisfiable formula is found. While an existing SAT solver could be employed for the task, we implemented from scratch the Davis-Putnam Procedure following the algorithm reported by Selman et at. [Sel96]. The rationale for the choice relies on one hand on the fact that SAT problems of the size from 4 to 40 variables and 17 to 172 clauses, as the developer is suggested to specify, can be easily solved by our in-house developed SAT solver. On the other hand we don't add a dependency on an external tool that would have made the deployment of the tool more complicated. Our version of the Davis-Putnam Procedure is implemented in the Python programming language.

2.7.2.3 Graph Generation

Once an unsatisfiable formula is found, the graph G prescribed by Karp's reduction to the k-clique problem is generated and its encoding as adjacency matrix is added to the output file.

By construction the matrix will have a size of $9M^2$ chars, where M is the number of clauses, i.e. M=floor(4.3*N).

2.7.2.4 Constant-generating Function Forging

The developer, using another annotation parameter, specifies the number NBITS of bits required to store the constant value. For each bit, a matrix is generated as described in the steps above. Figure 7 shows a snippet of the generating function. The first loop randomly generates a set of vertexes. The second loop verifies whether the subgraph induced by the set of vertex is a click. If it is not, which is always the case by construction, the bit is set to the required value. The chunk of code is repeated for all NBITS. Variable *res* collects the value of the constant which is returned on exiting the function.

Section 3 Task T2.2: White Box Cryptography

Section Authors: Brecht Wyseur, Patrick Hachemane (NAGRA)

3.1 Introduction

Task 2.2 is a task that runs for the first 2 years of the project, M1-M24. The activities of the first year have been reported in Deliverable D2.01 “Early White-Box Cryptography and Data Obfuscation Report”. D2.01 comprises an introduction to white-box cryptography with a brief overview of the state of the art, followed by a detailed description of the activities conducted in the first year. This included

- The design of the White-Box Tool for ASPIRE (WBTA) (Section 10 of D2.01), which was delivered as Release 1.00 in M12, and
- Research on new WBC schemes with provable security (Section 11 of D2.01).

In M18, an updated WBTA was delivered, tagged as Release 1.2.0, and reported upon in Deliverable D2.04 “White-Box Crypto Library and Code Generation”.

In this deliverable, we report the progress since D2.01 (M12) and D2.04 (M18). In particular, this captures the following progress:

- The design and implementation of a white-box AES (Advanced Encryption Standard) implementation, where the key is hardcoded into the source code that is generated by the White-Box Tools. This implementation is a step back from the provably white-box constructions in order to achieve performances that are acceptable for the ASPIRE use-cases, as foreseen in the DoW. Because of the trade-off between performance and security, we call this implementation **a time-limited white-box implementation**, i.e., a white-box implementation that should only be considered secure for a limited amount of time. We elaborate on this in more detail in Section 3.2.
- The design and implementation of a **dynamic-key white-box AES implementation**. This is an implementation where the key is not hardcoded into the source code at generation time, but where the implementation can be instantiated later-on by using an obfuscated (protected) key. We elaborate on this in more detail in Section 3.2.
- **Improvement on the White-Box Tools** to support these implementations. In particular, to support testing them, and to support dynamic-key white-box generation processes; these are complex because of the additional server-side function for protecting the key that needs to be handled with.

We also investigated how white-box cryptography can be used for **diversifying and hiding the VM bytecode** as has been developed in Task 2.3. In this investigation we followed different strategies

- To investigate how the white-box code generation tools could be used to diversify the VM bytecode. The white-box tools receive a seed as input that allows to generate seed-dependent diversified white-box implementations. We investigated how this approach could be used for generating diversified bytecode instances. Unfortunately, this is challenging to adopt in the current approach of how VM bytecode is translated using the cross translator. It would require significant modifications on the translator tools. Modifying the white-box tools to support this is not feasible, because the white-box tools cannot receive as input code definitions; the definition of the schemes that need to be generated are hard-coded into the white-box modules.
- To investigate how white-box can help to hide the VM bytecode. We concluded that a pragmatic solution requires two steps: (1) to implement an on-demand bytecode decryption scheme, which uses a cryptographic algorithm to decrypt the bytecode just

before it needs to be executed, and then (2) to white-box that cryptographic algorithm.

Beyond the initial scope as describe in the ASPIRE Description of Work (DoW), we also implemented **support for the XTS mode of operation**. We implemented this to support the SFNT use-case. This required additional white-box implementations (both encryption as decryption were needed for this) as well as additional support in the White-Box Tools.

The work of this task has been split according to the complementary expertise of partners involved in the collaboration. As expert on cryptography, at NAGRA, research was more focused in development of the part responsible for the generation of white-box cryptography code. As expert on source code transformation, at FBK research was devoted to the implementation of code rewriting transformations.

In particular, FBK extended the tool that was delivered at M12, to support the new dynamic-key white-box AES delivered at M24. This required:

- Adaptation of the content of the XML configuration file, that the source code analysis part fills to drive the execution of the white-box cryptography code generation algorithm. In fact dynamic-key white-box AES requires new data to be used during code generation, such as the value of the initialization vector, initially not included in the tool delivered at M12;
- Adaptation of the signature of the white-box function to call, that is changed after M12 due to new and different parameters to be passed to the dynamic-key variant;
- Adaptation of the source code transformation for the dynamic-key variant. In fact, on the static-key case, the variable holding the key value should be removed from the code. Conversely, on the dynamic-key case, the variable needs to remain in the code and accept the (always changing) value of the dynamic key.
- Emission of a detailed log file to document the code transformation performed by this step.

3.2 White Box Cryptography

We have implemented two families of white-box implementations: a fixed-key white-box AES implementation and a dynamic key white-box AES implementation. The AES cipher was selected because this was needed for the ASPIRE use-cases, as identified in Year 1 of the ASPIRE project.

A **fixed-key white-box AES implementation**, in decryption mode, was delivered on April 13, 2015 and validated on test cases in the white-box tools and integration in the ACTC. This delivery is a set of python scripts, which we denote as a “white-box module”. The scripts are invoked by the White-Box Tool for ASPIRE (WBTA) and receive as input a seed, the key that needs to be hardcoded into the implementation, and additional parameters that allow to tweak the generation process. The output of source code (C code and header code) which the WBTA parses into a C source code file that can be integrated in the application that needs to be protected.

When this is applied on two test programs that we implemented, this results into an increase in the application size of 167 KBytes.

In a later delivery, we also provided a white-box AES implementation module that generates the encryption mode.

A **dynamic-key white-box AES implementation**, in decryption mode, was delivered on July 16, 2015. This too concerns a set of python scripts, but in contrast to the fixed-key white-box implementation module, it does obviously not receive a key as input parameter. Instead, the white-box module will generate additional code that allows transforming a given key into an obfuscated key that the dynamic white-box implementation is able to parse.

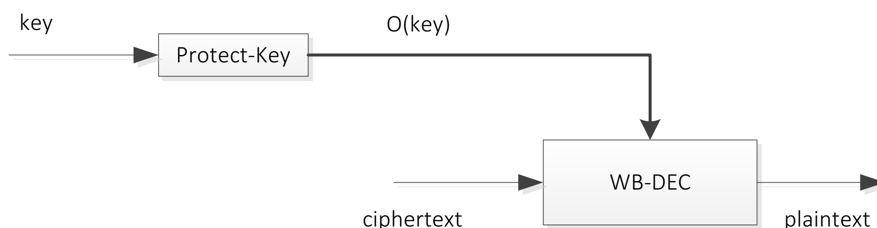

Figure 8 – Dynamic-key white-box high-level view

Figure 8 depicts the two functions that are generated and the dataflow between them. The function that protects a key receives as input the key by which the white-box implementation needs to be instantiated. This function should reside in a trusted environment like for example a trusted server. It produces an *obfuscated* key, which prevents the key itself from being recovered when recovered during transit or storage at client-side. The second function is the white-box descrambler itself, which has been generated as such that it can parse the obfuscated keys as such that the AES decryption operation with a key k is semantically equivalent to executing the white-box AES decryption function with the obfuscated key O(k).

In the white-box module that has been implemented, the protect-key operation turns a given 16 byte (128 bit) key into a 176 byte obfuscated key.

3.3 White Box Tool for ASPIRE

As explained in the previous section, white-box cryptography (WBC) is a particular implementation of a crypto algorithm that hides a key so that it is difficult to extract it, even with the source code at disposal.

Difficult does not mean impossible. Sooner or later, an attacker should be able to extract the key and access the secrets it protects. For this reason, WBC always should be used in combination with other protection techniques, like code obfuscation, anti-tampering techniques, etc. In addition, it should be diversified regarding:

- time: periodically, the implementation should be renewed;
- space: different implementations should be used for different products, segments, OS platforms or even single devices.

Renewability and diversity require a tool to generate the implementation, check its correctness and include it to the product. Therefore NAGRA proposed to develop an ad hoc tool named White-Box Tool for ASPIRE (WBTA).

3.3.1 Overview

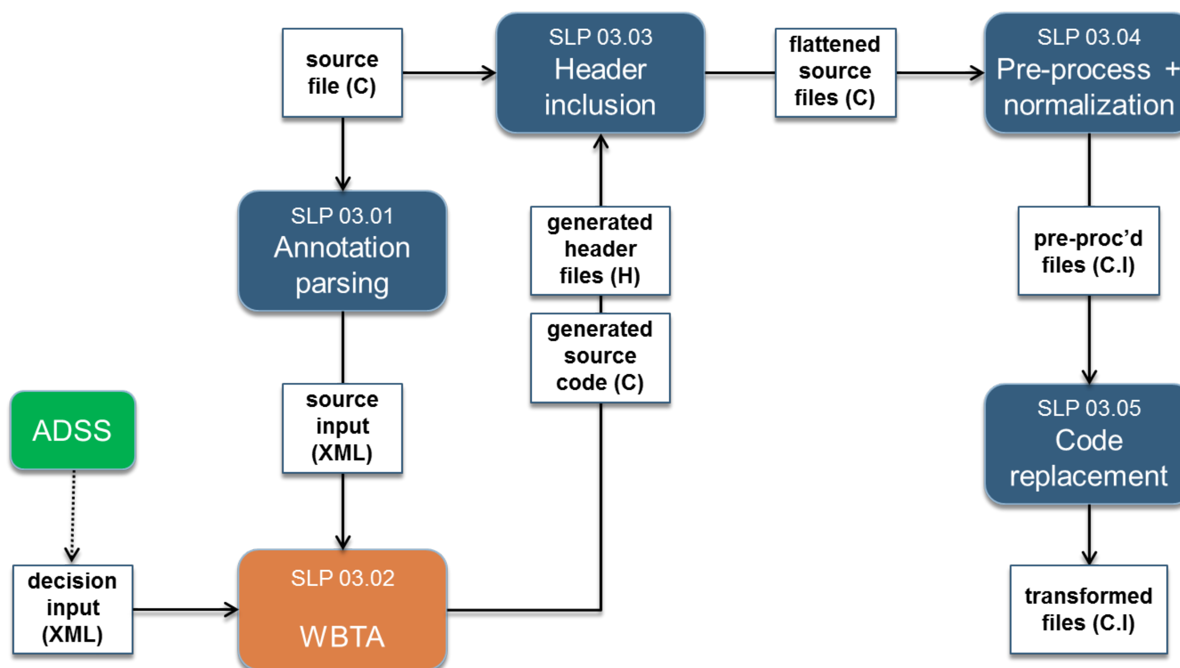

Figure 9 – T2.2 process overview

Figure 9 depicts an overview of the replacement of a standard (vanilla) cryptographic function call by a white-box cryptography (WBC) equivalent.

The vanilla function to replace is first annotated by the user. Then, the source file is used using the annotation parser (SLP 03.01), which results in an XML input file called source input. On the other side, in order to decide which primitive and which parameters must be selected to replace the code, a decision file is used. As of today, the file is hard-coded; it should be generated or fine-tuned by the ADSS in the next phase of ASPIRE.

Based on these two input files, WBTA generates the replacement code for the specified cryptographic function (SLP 03.02). The output is used as input of the header inclusion step (SLP 03.03), in order to flatten the source files. Next, the files are pre-processed and normalized (SLP 03.04); finally, the call to the vanilla cryptographic function is replaced by the one to the WBC primitive (SLP 03.05).

3.3.2 Technical choices

In order to ensure code portability, WBTA is written in Python 2.7, including support for Python 3.x. On the client side, the generated code (functions to process data using white-box crypto primitives) is in the C language; on the server side, the generated code (script to protect a dynamic key) is in Python 2.7. Input files are XML-formatted.

3.3.3 Previous steps

As described in document D2.01, WBTA 1.0.0 has been delivered to ASPIRE on 23 Oct 2014, with support for an XOR algorithm (a very lightweight form of encryption useful for tool demonstration only), with fixed key.

WBTA 1.1.0 has been delivered on 19 Feb 2015 (refer to document D2.04) and introduces the support of AES, DES and triple DES with fixed key; moreover, ECB, CBC and inverse CBC chaining modes are supported.

WBTA 1.2.0 has been delivered on 17 Apr 2015 (refer to document D2.04) and integrates a real primitive for fixed-key AES.

3.3.4 M24 achievements

3.3.4.1 Overview

WBTA 1.3.0 has been delivered on 29 Jul 2015 and was documented in the internal working document WD2.04b. It introduces the support of white-box cryptography applied to a dynamic key, also known as dynamic WBC. In such case, the key to protect is not static (fixed) in the application, but is dynamically transmitted during the execution of the application. This is typically the case of a *content key* used to descramble a video in a pay-TV system. In order to protect such a key, it is necessary to protect the original (vanilla) key on the server side, to transmit it protected, and to use a function on client side that decrypts the data using the protected key, without revealing the vanilla key. WBTA provides the two elements:

- a protection script, in Python, that protects the vanilla key;
- a code fragment, in C, that decrypts a data block given the encrypted data and the protected key as inputs.

Note that the encryption process does not change: data are encrypted like usual, using the vanilla key on server side.

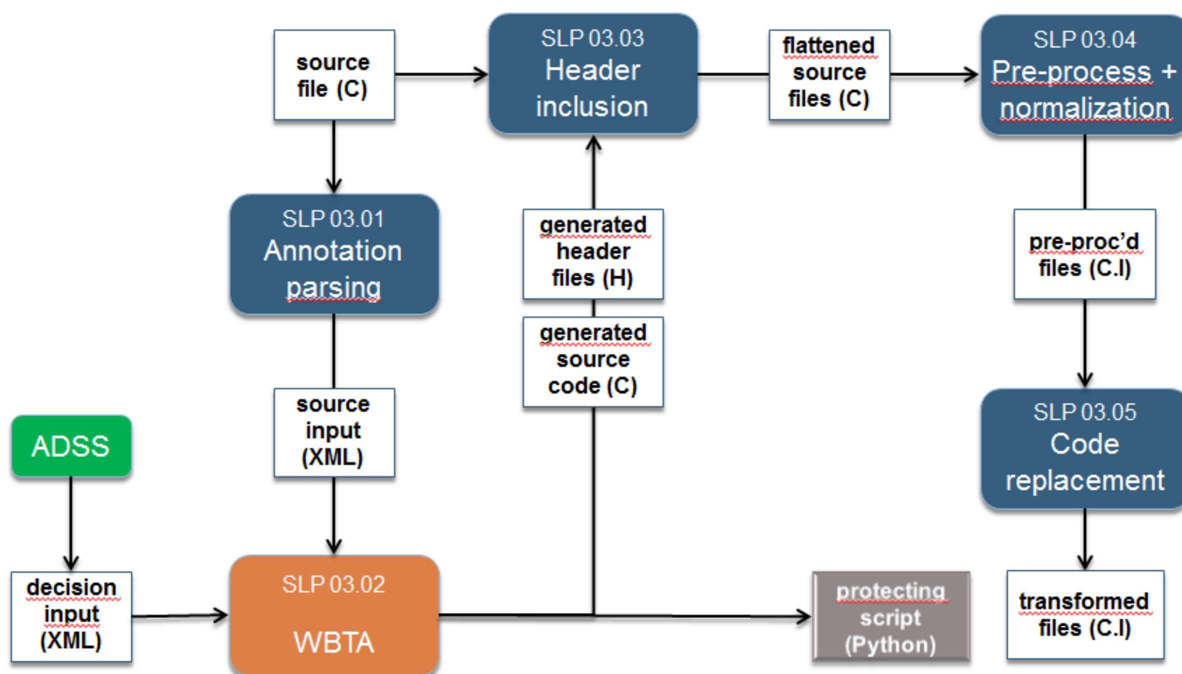

Figure 10 – T2.2 process overview - dynamic key

Figure 10 depicts the process. In comparison to Figure 9, WBTA provides the protecting script intended to be used on the server side to protect the key. This script is used on the server side to protect the key before delivering it to the client application, as shows Figure 11.

Next sections detail the improvements provided by release 1.3.0.

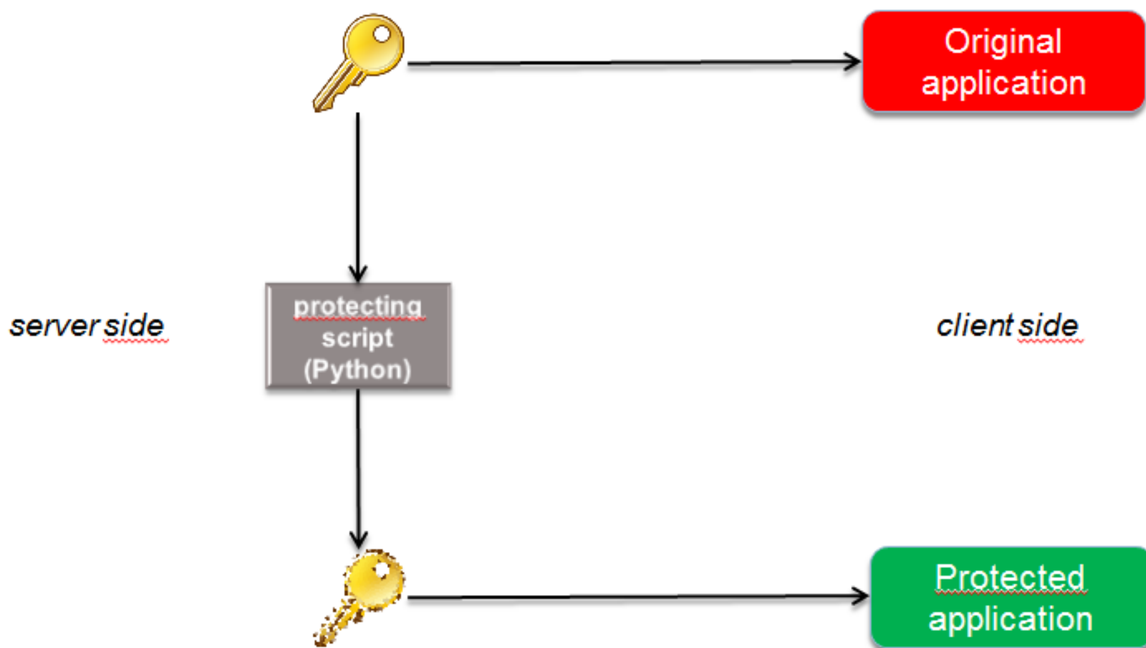

Figure 11 – Dynamic key protection

3.3.4.2 Input data and annotations

In a fixed-key implementation, the used key is hard-coded in the application and used during the call to the cryptographic function. During WBC replacement, the value of the key is removed from the code and somehow hidden in the WBC code. This means that the value of the key must be indicated in the annotation, so that WBTA can obtain it as input to the generated code.

In a dynamic-key implementation, the used key is dynamically delivered to the application. After WBC replacement, the key still is delivered dynamically, but in a protected form. As the size of the protected key may differ from the size of the vanilla one, this value must be specified as parameter, so that the generated code is able to use it.

These constraints triggered some modifications in the annotation format, as well in the annotation parser, the source input file and the WBTA itself. All these modifications have been introduced with support of backward compatibility.

3.3.4.3 Output data and integration to ACTC

As explained before, in dynamic case, WBTA produces an additional output: the script intended to protect the vanilla key on server side.

No changes were needed in the ACTC itself: this is because dynamic WBC is an offline protection. This means that the protection of the license key used on client side is done offline and this is out of the scope of the ACSL (ASPIRE Common Server Logic).

In case of the dynamic WBC, the protecting script is automatically generated during the phase SC 04.01 of the ACTC build process (see several deliverables D5.0x); it can be retrieved in the related directory of ACTC output.

3.3.4.4 Examples

WBTA is delivered along with a set of documented examples, also used to validate the tool at module level (module testing).

Initially, FBK provided a toy example checking the validity of a license.

The example has been enriched during previous phases, using different algorithms to encrypt the license data, with fixed or dynamic key (examples *license2*, *license3*, *license4*, *license5* and *license5b*).

Two additional examples have been delivered with the last release, specifically related to dynamic use case.

The first one is named *license6*. It is derived from the initial example *runLicense*, provided by FBK. The difference is that the key used to protect the license is encrypted with a random key on server side; then, both the encrypted and the random keys are transmitted to the client and used to decrypt the license. This is a typical case where dynamic WBC can be used to hide the value of the protecting key.

The second example, *license7*, is similar to license6, but the license key is protected using the decryption method of the crypto algorithm, on server side. This means that the dynamic WBC must hide the encryption method of the algorithm on client side.

In addition to these examples, two examples have been delivered to check the integration of WBTA and FBK tools with ACTC. The first one, *license_aes*, has been delivered during previous phases and is based on a fixed-key implementation. During this phase, the example *license_aes_dynamic* has been delivered along with JSON files used as input for ACTC, as well as the script `start_demo.sh`, that launches ACTC to compile the application, protects the keys using the generated Python script, and starts the application to check that the license can be decrypted using the protected key.

Section 4 Task T2.3: Client-Side Code Splitting

Section Authors: Bjorn De Sutter (UGent), Andreas Weber (SFNT)

4.1 Automated detection of code regions to split off

One work item of Task 2.1 as originally described in the project DoW consisted of profile-guided, multi-objective optimization techniques for program slicing to identify code regions to split off to protect variables, including source-level support. This work would build on the already available FBK-provided support for client-server code split point determination from task T3.1 in WP3.

Already in the first months of year 2 of the project, the consortium discussed this topic looking for more concrete approaches to start experimenting with, and realized that this topic would be very challenging.

Even before we raised this issue ourselves with the project advisory boards, the members of the board (and some other experts at the industrial partners not active within the project itself) anticipated this issue by pointing out that in general, we should not aim for tools that automatically detect which code to protect. The mentioned reasons for not doing so included

- the very application-specific nature of the relevant features of assets to protect and their relations, for which it is very hard if not impossible to develop a generic identification approach;
- the fact that aliasing hinders precise automated program analysis to such an extent that in practice the user would have to guide the tools anyway;
- the fact that the user is already annotating a lot of code, thus identifying it, by means of annotations anyway. It is consequently not much of a burden to require the user to explicitly identify all code that needs to be protected/transformed.

In summary, abandoning this work item would not endanger the practical usability of the ACTC, and hence not endanger the exploitation of the project results.

Furthermore, abandoning this work item for the client-side code splitting protection does not impact any other protection in the project. While several other protections will build on the client-side code splitting to implement advanced protection forms, none of those depend on the automated identification of code regions to be protected.

By contrast, abandoning this work item would free resources (at FBK) to spend on other work items considered more critical for the project.

For these reasons, we decided to directly follow the advice of the advisory boards and the external experts, and we decided to abandon this work item.

4.2 X-Translator:

The X-Translator that was provided as background to the project has in the meantime been enhanced independently of the project. In SFNT it is used as a proof of concept for experimenting in this direction. Enhancements are also re-used here inside the project. To support the project-specific adaptations had to be put in place and a test framework to support the quality level for ASPIRE had to be developed.

The further development focused on providing support for more complex code fragments, so that larger pieces of an ARM application can be translated into SoftVM (the interpreter embedded in the protected application) bytecode.

At M12 a translatable code fragment was fairly limited in its capabilities:

- It could only consist of a very limited subset of the ARM instruction set.
- It could not use control flow, only straight line code (a single basic block with a single exit) was supported.
- It is not possible to embed the continuation address into the bytecode but instead the VM invoking native code must provide this address at run time to the SoftVM.
- The code fragment could not access memory.

The following sections discuss the state at M24. While the work to achieve this state was done by SFNT outside the project and re-used by the project, we report it here to clarify that the goals foreseen in the DoW and the year two outlook presented during the first year technical review have indeed been reached at the end of year 2.

4.2.1 Supporting a larger subset of the ARM instruction set

Support for the following ARM instructions has been added:

- `bic reg, reg, imm`
- `bics reg, reg, imm`
- `bic reg, reg, reg, shift`
- `bics reg, reg, reg, shift`
- `clz reg, reg`
- `cmn reg, imm`
- `cmn reg, reg`
- `cmp reg, imm`
- `cmp reg, reg`
- `eor reg, reg, imm`
- `eors reg, reg, imm`
- `eor reg, reg, reg, shift`
- `eors reg, reg, reg, shift`
- `lsl reg, reg, imm`
- `lsls reg, reg, imm`
- `lsl reg, reg, reg`
- `lsls reg, reg, reg`
- `lsr reg, reg, imm`
- `lsrs reg, reg, imm`
- `lsr reg, reg, reg`
- `lsrs reg, reg, reg`
- `mla reg, reg, reg, reg`
- `mlas reg, reg, reg, reg`
- `mls reg, reg, reg, reg`
- `movt reg, imm`
- `mvn reg, imm`
- `mvns reg, imm`
- `mvn reg, reg, shift`
- `mvns reg, reg, shift`
- `rsb reg, reg, imm`
- `rsbs reg, reg, imm`
- `orr reg, reg, imm`
- `orrs reg, reg, imm`
- `orr reg, reg, reg, shift`
- `orrs reg, reg, reg, shift`
- `qadd reg, reg, reg`
- `qadd16 reg, reg, reg`
- `qadd8 reg, reg, reg`
- `qsub reg, reg, reg`

- qsub16 reg, reg, reg
- qsub8 reg, reg, reg
- qasx reg, reg, reg
- qsax reg, reg, reg
- qdadd reg, reg, reg
- qdsub reg, reg, reg
- rbit reg, reg
- rev reg, reg
- rev16 reg, reg
- revsh reg, reg
- ror reg, reg, imm
- rors reg, reg, imm
- ror reg, reg, reg
- rors reg, reg, reg
- rrx reg, reg
- rrxs reg, reg
- rsb reg, reg, reg
- rsbs reg, reg, reg
- rsc reg, reg, imm
- rscs reg, reg, imm
- rsc reg, reg, reg
- rscs reg, reg, reg
- sadd16 reg, reg, reg
- sadd8 reg, reg, reg
- sasx reg, reg, reg
- sbc reg, reg, imm
- sbcs reg, reg, imm
- sbc reg, reg, reg, shift
- sbcs reg, reg, reg, shift
- sbfx reg, reg, lsb, width
- sdiv reg, reg, reg
- sel reg, reg, reg
- shadd16 reg, reg, reg
- shadd8 reg, reg, reg
- shasx reg, reg, reg
- shsax reg, reg, reg
- shsub16 reg, reg, reg
- shsub8 reg, reg, reg
- smlabb reg, reg, reg, reg
- smlabt reg, reg, reg, reg
- smlatb reg, reg, reg, reg
- smlatt reg, reg, reg, reg
- smlad reg, reg, reg, reg
- smladx reg, reg, reg, reg
- smlal reg, reg, reg, reg
- smlalbb reg, reg, reg, reg
- smlalbt reg, reg, reg, reg
- smlaltb reg, reg, reg, reg
- smlaltt reg, reg, reg, reg
- smlald reg, reg, reg, reg
- smlaldx reg, reg, reg, reg
- smlawb reg, reg, reg, reg
- smlawt reg, reg, reg, reg

- `smlsd reg, reg, reg, reg`
- `smlsdx reg, reg, reg, reg`
- `smlsld reg, reg, reg, reg`
- `smlsldx reg, reg, reg, reg`
- `smmla reg, reg, reg, reg`
- `smmlar reg, reg, reg, reg`
- `smmls reg, reg, reg, reg`
- `smmlsr reg, reg, reg, reg`
- `smmul reg, reg, reg, reg`
- `smmulr reg, reg, reg, reg`
- `smuad reg, reg, reg`
- `smuadx reg, reg, reg`
- `smulbb reg, reg, reg`
- `smulbt reg, reg, reg`
- `smultb reg, reg, reg`
- `smultt reg, reg, reg`
- `smulwb reg, reg, reg`
- `smulwt reg, reg, reg`
- `smusd reg, reg, reg`
- `smusdx reg, reg, reg`
- `ssax reg, reg, reg`
- `ssub16 reg, reg, reg`
- `ssub8 reg, reg, reg`
- `sxtab reg, reg, reg, ror`
- `sxtab16 reg, reg, reg, ror`
- `sxtah reg,reg, reg, ror`
- `sxtb reg, reg, ror`
- `sxtb16 reg, reg, ror`
- `sxth reg, reg, ror`
- `teq reg, imm`
- `teq reg, reg, shift`
- `tst reg, imm`
- `tst reg, reg, shift`
- `uadd16 reg, reg, reg`
- `uadd8 reg, reg, reg`
- `uasx reg, reg, reg`
- `ubfx reg, reg, imm, imm`
- `uhadd16 reg, reg, reg`
- `uhadd8 reg, reg, reg`
- `uhasx reg, reg, reg`
- `uhsax reg, reg, reg`
- `uhsub16 reg, reg, reg`
- `uhsub8 reg, reg, reg`
- `umaal reg, reg, reg, reg`
- `umlal reg, reg, reg, reg`
- `umlals reg, reg, reg, reg`
- `sub reg, reg, imm`
- `subs reg, reg, imm`

4.2.2 Embedding the continuation address

At M12 binary code splitting did not yet embed the continuation address into the bytecode, but instead pushed it onto the stack prior to invoking the SoftVM. This provides the SoftVM with the information where the native execution should continue when leaving the bytecode.

Although easy to implement, this call-like scheme is fairly limited as it cannot support code fragments with multiple exit points, e.g. an if-statement, where the true path passes control to address 0xC001C0DE and the false path to address 0xBADC0DE. To support this kind of fragment the X-Translator was extended to be able to embed one or more continuation addresses inside the bytecode so that the bytecode image itself knows where native execution should continue.

A simulated protection workflow is used to test X-Translator/SoftVM features end to end on real ARM hardware without the need of prior ACTC integration. This is done by simulating Diablo's binary rewriting (which extracts native code from the binary in the ACTC) using different versions of handwritten ARM assembly.

With the above implementation a sample application can invoke the SoftVM by jumping to the associated native code stub. The SoftVM than interprets the VM image (at this stage still straight line code) and jumps back to the bytecode embedded continuation address.

4.2.3 Supporting shared objects

Support for shared objects has been added to the X-Translator & SoftVM already outside the project. This provides the baseline to support Android native support, as all Android native code is provided via shared objects.

With shared objects it is no longer possible to embed the continuation address as an absolute address into the bytecode, because it is not known at protection time but only at run time. To solve this, the symbols inside the JSON file that describes the extracted native code fragment to be translated by the X-translator no longer define absolute addresses but instead offsets from the shared object's base address. The base address is assigned by the dynamic linker when it loads a shared object into memory. The assembler glue code expects an absolute continuation address from the bytecode, so it is the bytecode's responsibility to calculate the address at run time by adding the embedded offset to the base address of the shared object. To do this the SoftVM interpreter (`vmExecute`) retrieves the base address from the dynamic linker and passes it to the bytecode as a part of the machine context.

4.2.4 Post-Linker interface

Embedding the continuation address inside the bytecode results in a chicken-egg problem: Before the bytecode can be generated the shared object's memory layout must have been fixed so that the final addresses are known. But finalizing the memory layout requires the bytecode images as these must be part of the shared object's memory image. To solve this problem the bytecode is generated twice. During the first generation the addresses are not known and instead dummy values are used. The purpose of the resulting bytecode is just to learn its size, so that the subsequent layout process can finalize the memory layout. Once the memory layout is fixed, the bytecode is generated again, this time using the real addresses instead of the dummy values. Afterwards the shared object is patched by replacing the placeholder bytecode from the first generation with the final bytecode from the second generation.

In ASPIRE the layout process and the creation of the final binary is done by Diablo. This means Diablo must be able to pass the symbol's final addresses to the X-Translator and also receive the final bytecode, so it can write it into the binary. Therefore the X-Translator's functionality was also made available as a shared object and a new function (`bin2vm_diablo_phase2`) was added, so Diablo can simply call the X-Translator to retrieve the final bytecode. This function accepts a buffer containing the JSON file content as a string and returns the generated bytecode as a linked list. It is expected that the passed JSON file defines addresses for all symbols. The order of the returned list follows the chunk definition from the JSON file. The list can be freed with the function `bin2vm_free_vmimages_arm`.

4.2.5 Chunk internal control flow

The now available chunk internal control flow was made available for the use inside ASPIRE.

4.2.6 Chunk test framework

To support the quality level required by ASPIRE the framework has been extended. The test framework from M12 includes semi-automatic generation of test case data to ensure the bytecode versions behave like real ARM hardware. Support for new instructions has been added to also assure good test coverage. Because that framework only targets individual ARM instructions, it was not suitable to assist development and quality assurance of chunk internal control flow, symbol support/embedded continuation address and multi-exit support. To also enable test driven development and automatic unit tests for these features the X-Translator has been extended to also support the definition and execution of test cases for complete chunks.

In this second test framework the test cases are defined inside a JSON file. So in addition to the basic blocks and edges a chunk optionally can also define a list of test cases. Each test case defines the input and the expected output values for the machine context. Any register not included in these lists will be set to zero prior to invoking the SoftVM and it is expected that the bytecode does not alter these unspecified registers.

A test case might look like this:

```
{ "input": {
    "cpsr": "0x00000000", //No flags.
    "r0":   "0xDEADBEEF",
    "r1":   "0xDEADBEEF",
    "r2":   "0x65",
    "returnAddress": "0x0" },
  "expected_output": {
    "cpsr": "0x60000000", //Z-flag, C-flag.
    "r0":   "0x13ba",
    "r1":   "0x65",
    "r2":   "0x65",
    "returnAddress": "0x1" }
}
```

To enable execution and verification of these test cases the X-Translator's post-linker interface was extended, so that `bin2vm_diablo_phase2` does not only return a list of bytecode images for the given JSON file, but that each bytecode image is optionally accompanied by a list of test cases. Each test case contains a machine context with the input values and another machine context with the expected output values.

The X-Translator was modified, so it supports a new test mode, which can be activated with the command line switch `--phase 3`. In this testmode the X-Translator first translates the JSON file using the post-linker interface and then additionally executes each bytecode image with all its test cases in a SoftVM where it checks if each test case invocation produces its expected values.

4.2.7 Integrating the LLVM Interpreter (lli)

In parallel to ASPIRE, SFNT modified the original SoftVM interpreter to leverage the lli (LLVM Interpreter) as an execution engine. This supports easier diversification of the bytecode. In ASPIRE, the improved lli-based SoftVM is now used as well.

4.2.8 Supporting memory access

Enabling for ASPIRE the use of the new capability of the SoftVM to access memory of the application required adaptations of the X-Translator.

- Add support for an artificial `address_producer` instruction that loads a register with the absolute address of a named memory location.
- Extend the chunk test framework with the ability to define and verify test cases involving memory.
- Translate ARM's various load and store instructions into equivalent LLVM-IR.

In addition the Instruction-Selector Interface was extended to deliver the correct information to Diablo.

An `address_producer` is an artificial instruction that behaves like a `mov reg, imm` where the immediate can occupy 32bits and corresponds to a symbol value. This enables the bytecode to address arbitrary memory locations inside the ASPIRE-protected shared object. Such an instruction cannot be natively available on ARM because every instruction has a length of 4 bytes (32bits) making it impossible to fit an opcode and a 32bit immediate into one instruction. Actual ARM code uses a variety of code patterns to achieve the same result, e.g. by separately setting the lower and higher 16bit of the target register or using a program-counter relative load from a reachable constant pool. It is the responsibility of the chunk extractor (which is Diablo in ASPIRE) to recognize these patterns and canonise them into appropriate address producers.

Inside the JSON file such an address producer and its corresponding symbol might look like this:
Address producer:

```
{"type": "address_producer", "addrsymbol": 7, "addrregister": "r4"}
```

Symbol #7:

```
{"name": "a_variable", "address": "0x1234"}
```

The X-Translator translates this definition into bytecode that loads the absolute address of "`a_variable`" into register `r4` by adding `0x1234` to the shared object's run-time base address and storing the result into register `r4`. It is the responsibility of the post-linker (in ASPIRE: Diablo) to generate the JSON file with correct symbol offsets.

For an adequate level of quality, it is important to extend the existing chunk test framework, so it also supports automatic verification of memory test cases.

The extended test framework allows the definition of memory regions inside the JSON file. These memory regions only have a meaning for testing and define a length and an associated symbol.

An example memory region and its associated symbol:

Memory region #3:

```
{ "symbol": 7, "size": "4" }
```

Symbol #7:

```
{ "name": "a_variable", "address": "0x1234" }
```

The test case definition in the JSON file had also been extended, so one can define memory pre and post images. This might look like this:

Pre image:

```
{ "mem_id": 3, "image": "0x01000000" }
```

A matching post image:

```
{ "mem_id": 3, "image": "0x10000000" }
```

The extended test case includes the 4 byte variable `a_variable`, which lives in memory. Before executing the test case the framework initializes the variable with the integer 1 (`0x01000000` in little endian) and after executing the chunk under test, it will check if the variable had been updated by the chunk to contain the value 16 (`0x10000000` in little endian).

Supporting these memory test cases requires additional support in the X-Translator, because the bytecode contains hardcoded memory addresses. Therefore the X-Translator learned a new test mode (command line switch `--phase 3`) that behaves very similar to the bytecode generation during the post-linking step (`bin2vm_diablo_phase2`). The difference is that this mode does not hardcode the memory addresses as defined by the JSON file but instead dynamically allocates the memory regions and uses the addresses returned by `malloc` for the corresponding symbols. This way the generated bytecode is tailored towards X-Translator's own address space allowing its execution by an embedded SoftVM. The bytecode generation does not only return the bytecode for each chunk, but also a list of test cases. In addition to the register input and expected output values, each test case also carries a list of memory behaviours. A memory behaviour specifies the expected behaviour of a memory region by containing a pointer to the region and its pre and post image.

When running in test mode the X-Translator verifies each chunk against all its test cases. To verify a test case the X-Translator first initializes the registers with the input values and the memory regions with the pre images and then executes the chunk with the embedded SoftVM. Once the execution finished it checks if the actual output (register values and memory content) matches the expected output as defined by the expected register values and the post images.

With the ability to define unit tests for memory operations it was fairly straight forward to implement and verify the translations for ARM's load and store instructions. Currently the X-Translator supports the following load/stores:

- `ldr reg, [reg]`
- `str reg, [reg]`
- `ldr reg, [reg, imm]`
- `str reg, [reg, imm]`
- `ldria<!> reg, { reg_list }`
- `stria<!> reg, { reg_list }`
- `strda<!> reg, { reg_list }`
- `ldrda<!> reg, { reg_list }`
- `strdb<!> reg, { reg_list }`
- `ldrdb<!> reg, { reg_list }`
- `ldrib<!> reg, { reg_list }`
- `strib<!> reg, { reg_list }`
- `push { reg_list }`
- `pop  { reg_list }`

4.2.9 Tool versioning and automated release builds

After initial integration of the X-Translator into the ACTC, various bugs had been discovered requiring bug fixes. This lead to a relatively high release frequency, which demonstrated the need for strict versioning, automated packaging and automated regression testing.

To support strict versioning the ability to return a version string was added to each binary component of an X-Translator release build. Currently an X-Translator release includes the following binary components:

- `xtranslator`: Executable that provides X-Translator's command line interface.
- `libbin2vm.so`: Library that provides the Instruction-Selection and Post-Linker APIs and implements the actual translation from JSON to bytecode.
- `libsoftvm.so`: SafeNet's traditional stack based SoftVM built as an x86 library, used by `xtranslator` to execute test cases.
- `libwandivm.so`: The LLVM-based SoftVM built as an x86 library, used by `xtranslator` to execute test cases.

The version string consists of the following elements:

- Git tag: The name of the release, e.g. `EU_RELEASE_3.1.3`. Development builds are not tagged and identify themselves with an empty string.
- Build machine: Username and hostname of the machine that performed the build. Official releases identify themselves with `aspire@aspirevm`.
- Git hash: The source tree's commit-id the build was created from.
- Configuration: The build configuration, e.g. verbose/non-verbose, release/debug.

The build scripts had been modified to collect this information and make it available to the code via defines. The X-Translator learned the new command line switch `--version`, which prints the version strings of the components.

To support automated packaging an additional integration script had been added, that automatically creates a releasable archive from source. This script performs following tasks:

- Build 3rd party libraries (LLVM, Capstone, and YAJL).
- Build a non-verbose version of the X-Translator binary components.
- Build a verbose version of the X-Translator binary components.
- Run the X-Translator unit tests:
    - Instruction level unit tests for the traditional SoftVM.
    - Instruction level unit tests for the LLVM-based SoftVM.
    - Chunk level unit tests for the traditional SoftVM.
    - Chunk level unit tests for the LLVM-based SoftVM.
    - Memory access unit tests for the LLVM-based SoftVM.
- Collect the contents of the release archive:
    - X-Translator non-verbose build.
    - X-Translator verbose build.
    - Header files for the Instruction-Selection and Post-Linker APIs.
    - Source code and build script of the traditional SoftVM.
    - Source code and build script of the LLVM-based SoftVM.
    - A sample that demonstrates the usage of the X-Translator end to end on a simple ARM program with the traditional and the LLVM-based SoftVM.
- Create the release archive.
- Build the ARM executables of a special unit test for the traditional and the LLVM-based SoftVM. This unit test is dedicated to the generated glue code and checks if the information flow between native ARM and SoftVM is working in both directions. This makes sure the SoftVM actually receives the values of the physical ARM registers and that its calculated values correctly end up in the physical ARM registers.

With the above automation in place, delivering a new X-Translator release is a four step process:

1. Use git to tag the desired version of the X-Translator source tree (usually `HEAD`) with the release version, e.g. `EU_RELEASE_3.1.3`.
2. On the ASPIRE-VM: Use git to checkout the desired version of the source tree; run the integration script to create the release archive.
3. Upload the two ARM executables of the glue code unit test to an ARM development board; execute them and verify that they don't report an error. (At the moment this step is not automated because in the current setup the ASPIRE-VM is not on the same network as the ARM development board.)
4. Ship the release package (e.g. `EU_RELEASE_3.1.3.tar.xz`) to Gent University.

Client side code splitting only works reliably if X-Translator and Diablo agree on the same APIs. To ensure the ACTC uses compatible versions of Diablo and X-Translator, each X-Translator release is first sent to Gent where it is tested with the latest Diablo. After successful verification Gent updates the ACTC by replacing Diablo and X-Translator together.

Section 5 Task T2.4: Binary Code Obfuscation

5.1 Control Flow Obfuscation

Section Authors: Bjorn De Sutter, Bart Coppens (UGent)

The basic control flow obfuscation research was advanced in the project, and many results were hence already reported in D2.06 at M17

Additional research was conducted along two lines of research, which are documented in the next two sections.

5.1.1 Improving the deployment of existing obfuscations

To improve the deployment of the already implemented support for opaque predicates, control flow flattening, and brach functions, we worked on two **implementation** aspects.

Foremost, we extended how the obfuscations are applied to the code: whereas in our initial implementation, obfuscation transformations were applied stochastically, we now support the profile-guided application of obfuscations. In this mode, the obfuscator will focus on infrequently executed program points to insert the obfuscating instruction sequences, such that the run-time overhead is minimized.

Figure 12 shows the overhead of applying the branch function insertion and opaque predicate insertion obfuscations, using either the original, stochastic method or the profile-guided method (where the X percent least frequently executed blocks are selected per function) on the bzip2 SPEC2006 benchmark. The x-axis indicates the percentage of transformed code blocks, and the y-axis the execution time overhead. As can be seen, the stochastic method already introduces an overhead when only 10% of the blocks are transformed, compared to 0% overhead from the profile-guided approach. The profile-guided approach consistently produces less overhead, except for when all code blocks are obfuscated, in which case both methods produce the same result.

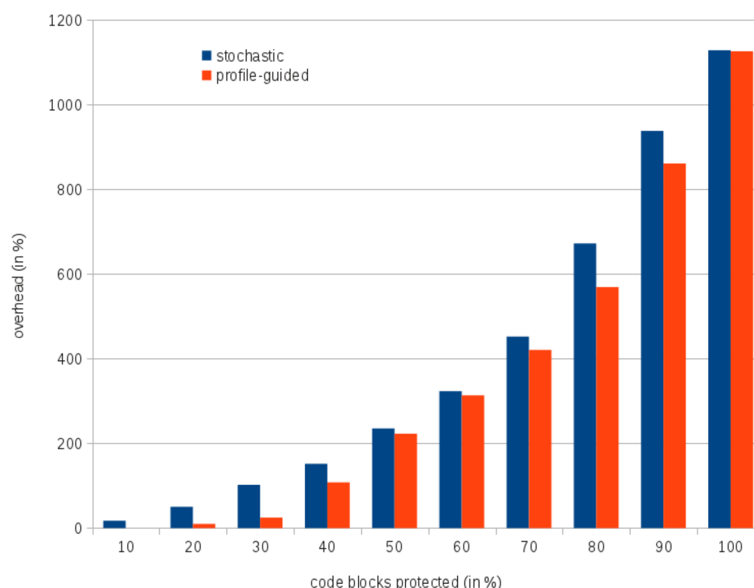

Figure 12 – Overhead comparison between stochastic and profile-guided obfuscation

Secondly, we debugged the implementation of the existing obfuscations. In particular, we developed the necessary IR (internal representation) bookkeeping functionality to maintain

all IRs in a fully coherent state. Before we developed this functionality, the obfuscations in Diablo could only be applied as the very last protection step in Diablo: The IR data structures was partially broken as a result of their application, which blocked the execution of new data flow analyses as needed to support additional protections later on in the execution of the obfuscator. In the now extended version, all IR data is correct and complete enough to support advanced later transformations, such as those needed for the code mobility protection developed in WP3.

All of these improvements are delivered as part of prototype deliverable D2.07.

5.1.2 Flexible, two-way opaque predicates

In D3.04, we report the research performed at UGent into delay data structures, i.e., data structures that can covertly store the results of attestations & attestation verifications to hide the direct link between a failed attestation/verification, and the triggered reaction.

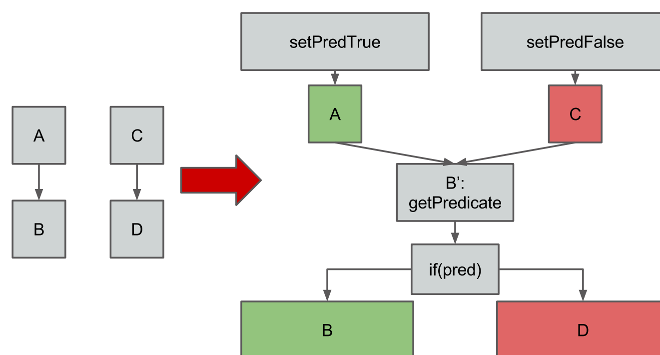

Figure 13 – Principle of flexible two-way opaque predicates

For the goal of improving binary control flow obfuscation, we researched the use of the same data structures for flexible two-way opaque predicates.

Figure 13 visualizes the principle of such two-way predicates. On the left, two (independent) code fragments from the program's control flow graph are depicted. On the right, the transformed fragments are shown. The red and green mark the basic blocks belonging together in the original code. But from the restructured control flow graph, this relation is no longer apparent: in the restructured graph, all blocks are connected to all blocks.

The two primitives on top of the restructured graph "setPredTrue" and "sedPredFalse" denote invocations of status-setting functions of a flexible data structure API, the "getPredicate" denotes an invocation of the a status-querying function of the API. For more details about those APIs, we refer to D3.04 Section 5.1.5. Here, the point is that those data structures are defined outside the obfuscator. They are defined by the user of the obfuscator which gives that user much more flexibility in choosing different data structures than when only built-in data structures of the obfuscator could be used.

By using those APIs and data structures, that are not known by an attacker in advance because they are not limited to a list of builtin data structures, the obfuscator can hide the relatively simple nature of this protection. The fact that the conditional branch based on the predicate will evaluate in both directions during a program's execution also ensures that the protection will withstand dynamic attacks that eliminate conditional branches of which the attacker observed that they evaluate to only one direction during execution on representative inputs. Such attacks can easily break static opaque predicates, but not our flexible two-way predicates.

Moreover, nothing prevents the user of the obfuscator to instantiate the flexible data structures by means of data structures already present in the program to be protected. Instead, the user is adviced to so. In that case, the functions invoked to implement the

setPredTrue, sedPredFalse, and getPredicate primitives will also be invoked as part of the normal execution of the program. So the semantics of those functions now become part of the original program semantics, as well as of the protections' semantics. Their run-time behavior will therefore feature less invariants, and will hence be harder to comprehend, and it will become harder for the attacker or for a de-obfuscation tool to separate the application code from the protection code, and to abstract away from the protected code. In other words, it will become harder to undo the protection.

The need for this form of protection, as an extension of the binary code obfuscations already mentioned in the project description of work, was inspired by a dynamic attack proposed by Saumya Debray at all in 2014, and published more extensively in 2015 [Yad15]. In this way, the project adapts to evolutions in the never ending arms race between offensive and defensive security techniques.

To inject flexible two-way opaque predicate, Diablo performs the following steps:

- Choose one of the user-defined predicates of the provide data structure.
- Choose 2 code blocks in the program's control flow that will be linked with the two-way opaque predicate.
- Both code blocks are split into two parts: each now consists of a predecessor and a successor.
- A new code fragment is injected, which contains a call to query the state of the predicate, and a conditional jump that depends on the result of this query.
- A call to a setter-function of the predicate is injected at the end of each predecessor block. The arguments are automatically chosen such that both calls set the predicate to the opposite value.
- Control flow is redirected from the end of each predecessor block, i.e., after the call to the setter-function, to the new block containing the call to the query function. The outgoing edges from the conditional jump in this block are directed to the successor blocks. These edges are added in such a way that this block redirects the control flow to the correct successor block for each of the predecessor blocks, depending on the predicate value.

Control flow is redirected from the end of each predecessor block, i.e., after the call to the setter-function, to the new block containing the call to the query function. The outgoing edges from the conditional jump in this block are directed to the successor blocks. These edges are added in such a way that this block redirects the control flow to the correct successor block for each of the predecessor blocks, depending on the predicate value.

Although this line of research is not yet finalized, and our experience with it is hence still immature, we can already report some evaluation results.

For this research, we used two metrics to check the cost of this transformation. We used size increase of the program after transformation and the increase in execution time. We have implemented the transformation for ARMv7 and executed the code on a development board which has 1GB DDR3 RAM and a quad-core ARM Cortex A9-processor. The OS running on the development board is Linaro 13.08, a Linux distribution.

Benchmarks

To test the overhead of the obfuscation, we transformed libquantum, bzip2 and Helloworld. Libquantum and bzip2 are two benchmarks of the SPEC2006 benchmark suite.

We tested the transformation using 3 data structures: 2 different implemenations of a linked list and quantum_reg, a data structure declared in the libquantum source code. We used two implementations of a linked list to get an overview on the impact of the implementation of the data structure. The functions, which change the value of the predicates in the first linked list, allocate and free a lot of memory. The functions of the second linked list only change integer values. We assume the transformation using the first linked list will slowdown the pro- gram

more than the second linked list. We reused the function quantum_addscratch to put the predicate encoded in quantum_reg on true. We implemented the function to put the predicate on false by ourself.

We used the linked lists to obfuscate all 3 programs. We only used quantum_reg on libquantum because we reused a part of the code from the original libquantum binary to set the value of the predicate.

Table 4 shows for each of the combinations of benchmark and data structure the number of predicates that was inserted in the program.

Benchmark	Data structure	Nr. Of inserted predicates
Helloworld	LinkedList1	13
	LinkedList2	13
Bzip2	LinkedList1	737
	LinkedList2	726
Libquantum	LinkedList1	223
	LinkedList2	217
	Quantum_reg	172

Table 4 – Overview of the benchmarks use to evaluate flexible opaque predicates

Program size

Figure 14 visualizes the increase of the program size for each data structure and benchmark. We can conclude the program size increases for all benchmarks. This is due to the fact we link extra code in the binary and add some extra instructions to call the functions which evaluate the predicate and change the value of the predicates.

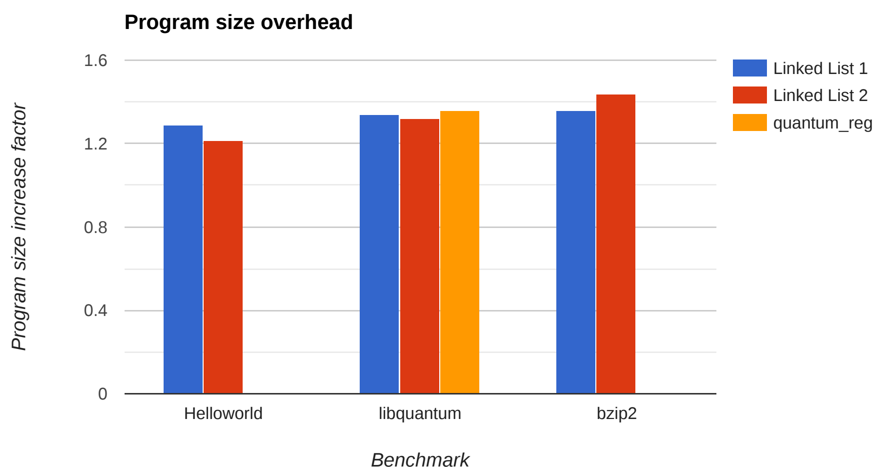

Figure 14 – Program size overhead of using flexible two-way opaque predicates.

Execution time

To measure the overhead of the execution time, we measured the original execution time and the execution time of the transformed program. We made a comparison between those timings and visualized these in Figure 14.

In this chart we see the number of times the execution time increases for each benchmark and data structure. We conclude that quantum reg introduces the most overhead of all data structures. This is due to the fact the function which puts the predicate on true, is large and complex. The functions which change the value of the predicates in the linked list, are smaller and less complex.

We can also conclude that the second linked list obfuscation slows down the program less than the first linked list. We can assign the extra cost for the first linked list to the implementation of the predicates: allocation and freeing memory are time-consuming.

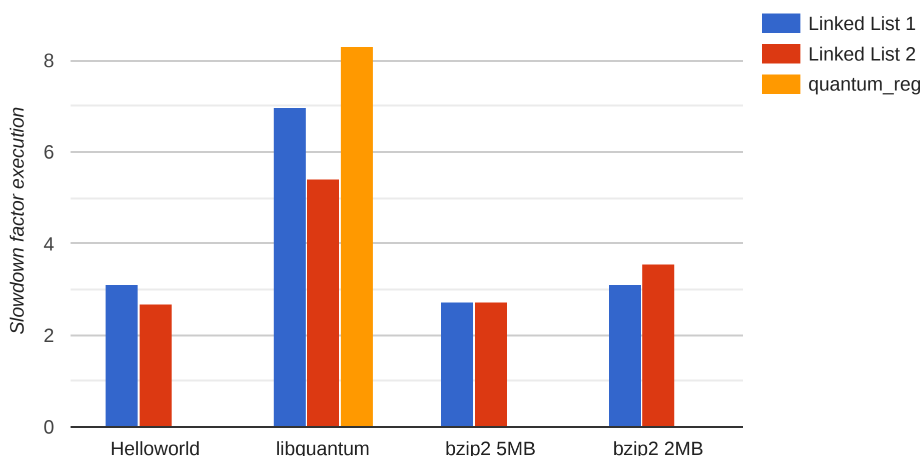

Figure 15 – Execution times for flexible two-way opaque predicates

A more extensive evaluation, and variations on the two-way predicate scheme are discussed in Thomas Van Cleemput's master thesis [Cle15].

5.2 Multithreaded Cryptography

Section Authors: Jerome D'Annoville (GTO)

Applications that need to exchange data in a secure way with a server need to embed a secret to set a secure communication with the application server. In symmetrical cryptography a master key can be deployed with the application. A key derivation function is used to generate a dedicated key derived from the master key that is later used to protect a device dedicated data. An advantage is that the same application can be deployed on all devices. The constraint is that a master key is hidden somewhere in the application and can be hacked by an attacker.

This Multi-threaded Cryptography protection proposes to prevent the exposure of a master key in an application by moving the key derivation operation onto a server that is called hereafter the Crypto server. Several derived keys are returned by the Crypto server that are used in parallel in the application to protect data. Among these keys, only one is the valid key. The recipient of the protect data is able to retrieve the valid data because he is able to derive the valid key since he is sharing with the Crypto server the way to derive the valid key.

The parallel processing and this overall Multi-threaded protection is not the topmost security protection. This is typically security provided by complexity and there is no ambition to block a determined and patient attacker. The purpose is to prevent an attacker to easily connect a

key with the cyphered text produced by the protection and then to determine which is the valid cryptogram.

5.2.1 Original AES

The AES cipher standard is originally meant to encrypt 128 bits blocks with a 128, 192 or 256 bits keys. The encryption is composed of several steps, based on highly non-linear permutations of the 16 bytes of the text and XOR operations. The structure of AES is based on round: it repeats the same operation a certain number of times with substitution-permutation at each round. There are 10, 12 or 14 rounds according to the key length. There are nice animations available on Internet such as [Abi11] that explains how AES works.

When the length of plaintext to cipher is not a multiple of 128 bits, it is necessary to pad the plain text. The padding standard chosen here is Public Key Cryptography Standard (PKCS) #7 [Kal98].

When several blocks have to be encrypted the mode of operation used is the Cipher Block Chaining structure (CBC). It creates dependencies between the encrypted blocks and creates some visible randomness: the encrypted block n-1 is used as the initialization vector for the encryption of the block n.

The Multi-threaded protection is based on a modified Advanced Encryption Standard (AES) encryption. For the purpose of the project the AES implemented in OpenSSL (https://www.openssl.org/) has been used.

5.2.2 Master key

The master key is still embedded in the application but it is ciphered to prevent an attacker to use or disclose it. The master key is encrypted with the public key of the Crypto server. Only the Crypto server owns the corresponding private key and then the master key is no more exposed in the application. A 2048-bit RSA key is used, note that the device only keeps the ciphered value but do not run the RSA decryption. This is done on the server side only.

5.2.3 Architecture

The new component introduced with this protection is the Crypto Server. The original call to standard crypto library in the application is replaced by a call to the crypto server and a call to the CryptoMultiThreaded library. As shown in the Figure 16the Derived keys are generated by the Crypto Server and are returned to the Application that calls the CryptoMultiThreaded to cypher the plain text (PT) in to several ciphered texts (CT).

The seed that is returned by the Crypto Server enables to retrieve the valid cypher text among the set of cipher texts passed to the Application server. The same seed is used to determine the permutation rule during the parallel AES rounds processing.

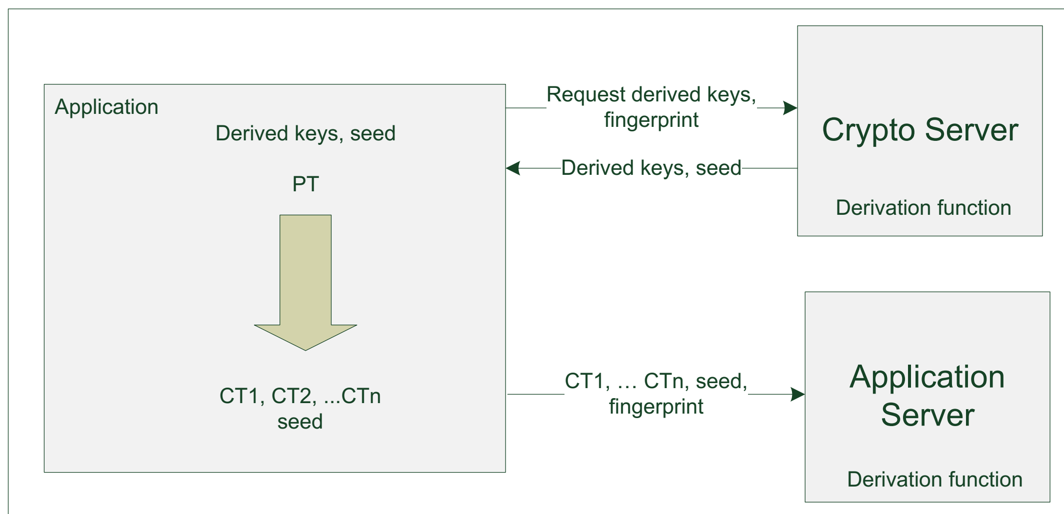

Figure 16 – Architecture of the Multi-Threaded Crypto protection

The Crypto Server and the application server shares the master key, the Derivation function, and the number of iterations of the Derivation function.

The Pseudo Random Number Generator (PRNG) that is used to determine the permutation during the obfuscation in the CryptoMultiThread library must be the same as in the Application server.

At the beginning of the exchange between the application and the applicative server, the application requests keys to the cryptographic server to perform the encryption with. The application gives to the cryptographic server

5.2.3.1 Crypto server

The purpose of the Crypto server is to generate a seed and several derived keys from a master key given as input. This server provides a simple service that could be generic: there is no specific application data to maintain over sessions and required arguments are provided as input by the calling application. The key derivation process is shared by the Crypto server and the Application server. Then a contract has to be set to enable to retrieve the same derived key on both servers. Then some configuration data are required that are application specific:

- The derivation function that is used can be specific for an application
- The hashing function used by the derivation must be determined as well for an application
- The iteration number used in the derivation process must be configured for an application

For the project the derivation function, the hashing function and the iteration number are fixed in order to simplify the implementation of the crypto server then no configuration is required

The Key derivation Password-Based Key Derivation Function 2 (PBKDF2) [Kal00] is used as derivation function.

The seed is randomly generated.

The way to set the position of the valid key must depend on something that is unknown on the client side and that changes for each occurrence of deployed application. The master key is ciphered in the application and cannot be retrieved by the attacker. The seed is generated on the Crypto server side for each installed application. The function used by the Crypto server to determine the position of the valid key in the set of keys returned to the application depends of the master key and the seed:

F (master_key, seed, number_of_keys) = (master_key ⊕ seed) % number_of_keys

The number of keys is fixed for the project. This is the level of parallelism of the AES, check the 5.2.6 paragraph on performances about this.

5.2.4 CryptoMultiThread library

As introduced in the architecture paragraph, the call to AES encryption is replace by a call to the crypto server and then a call to a modified AES encryption. In this modified encryption each round of AES is performed in several threads in parallel with a different key.

It is as if Nth AES are performed on the same plain text in parallel with different encryption keys except that at each round of the algorithm the data are permuted between the threads. The "data" mentioned here is an abstract shortcut to designate the states and the round keys. This permutation is an extra step done at each round of the AES algorithm as it is shown in the following figures.

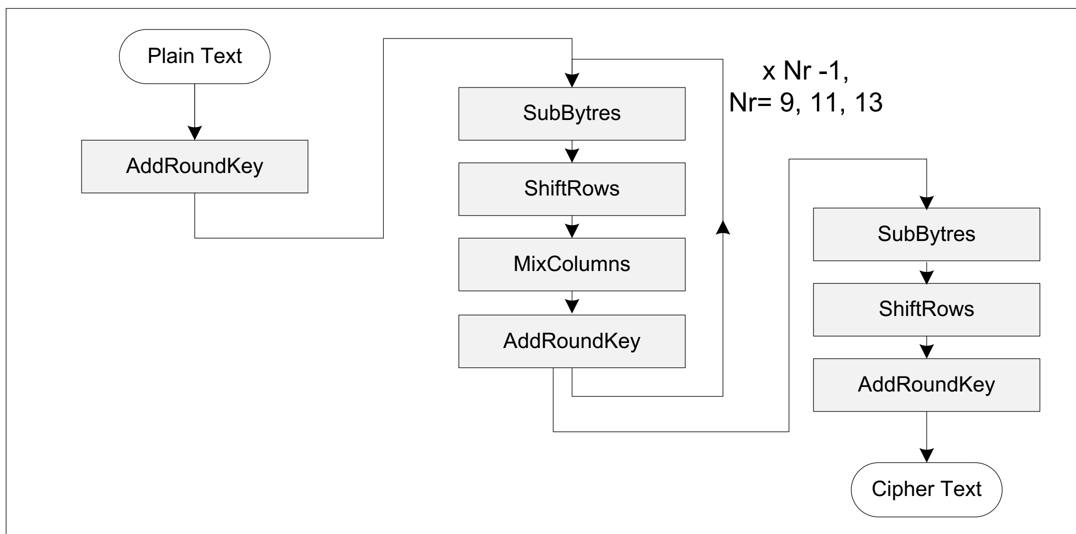

Figure 17 – AES

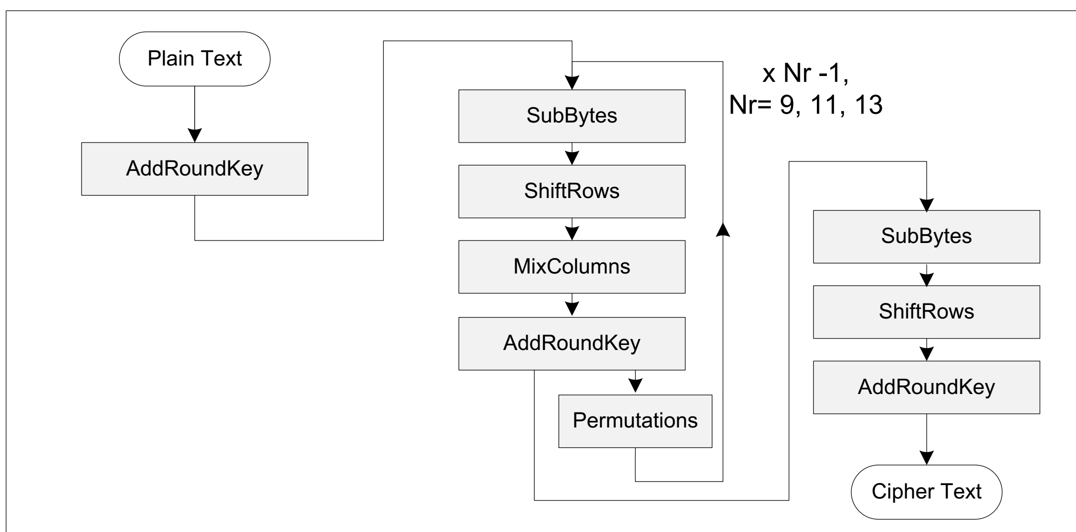

Figure 18 – Modified AES

The standard AES is depicted on Figure 18 and the on modified AES is on Figure 18. The algorithm remains unchanged with the four transformations (SubBytes, ShiftRows, MixColumns, AddRoundKey). An extra permutation step is done after each round as it is shown. A monitor synchronizes all threads that are performing the rounds. When all parallel rounds have been executed in their thread then the monitor modifies the data handled by each thread.

The order of permutation is provided by a PRNG that is initialized with the seed at the beginning. The permutation rule can be seen as a two dimension array that gives how the data are permuted among the threads. After each round data of a thread are assigned to another thread

Current assignment	*5*	*2*	*4*	*6*	*3*	*1*
Next assignment	*3*	*6*	*1*	*2*	*5*	*4*

The data processed by the thread 5 for the current round will be processed by the thread 3 for the next round. This can also be depicted with a diagram as in Figure 19 – Permutation rule.

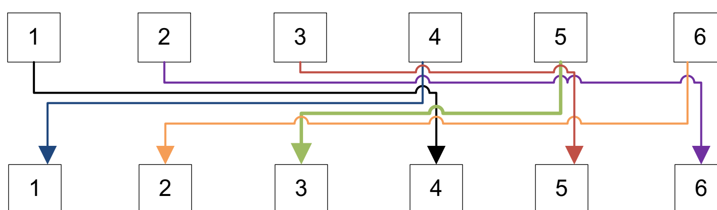

Figure 19 – Permutation rule

These permutation rules are generated with the Mersenne Twister PRNG. The TinyMT code [Mut11] is used because its small size it is adequate on an embedded device.

5.2.5 Application Server

The Application server shares the way to derive the key from the master key with the crypto server. The master key is kept on the Application server, all derivation required data are known by the Application server except the seed and the fingerprint that are passed by the application. Then the position of the valid cipher text must be deduced from the path that starts with the position of the valid key to the result has it is done on the client side. To ba able to do this it must use the same PRGN than the CryptoMultiThread. The same TinyMT code as describe previously is used on the Application server side.

5.2.6 Application Performance degradation

There is a serious degradation according the parallelism level as expected. The Figure 20 below shows that an important number of threads would significantly affects the performance of the application. Measures have been done on a Nexus 5 device.

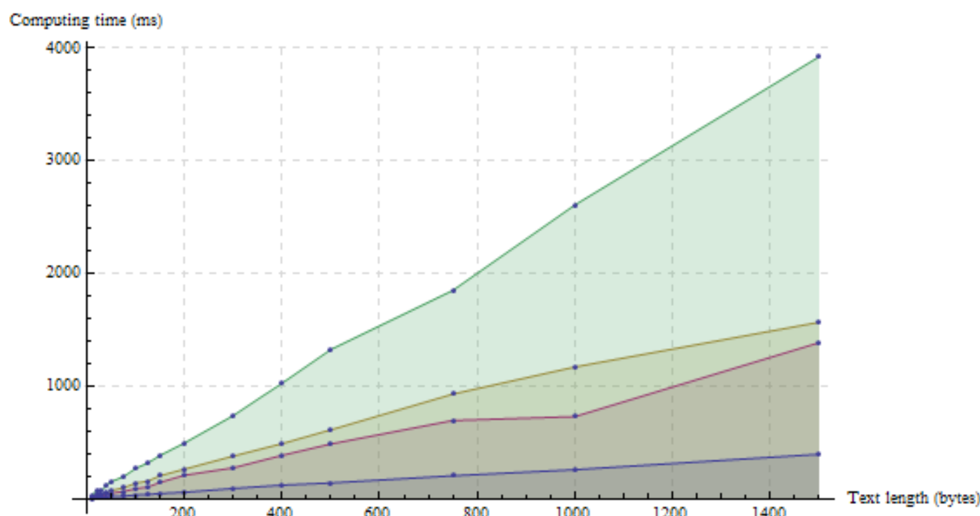

Figure 20 – Processing time with 128bit key for plain text sizes for 1, 3, 4 and 7 threads

Some other measures done on a Samsung S3 that in order to stay under one second processing for this ciphering task only the number of thread should be less than 5 for a 2048 bits message length.

For the purpose of the project the number of thread will be set to 4.

5.2.7 Limitations

The limitations of this protections are first that a very specific use-case is covered. The recipient should be a remote server, the messages should be limited in size otherwise the computing time penalty would be unacceptable.

Another very serious limitation is the fact that the application is impacted by the protection. The original call to the encryption can be automatically replaced by ACTC but since the size of the result has changed and that some additional data have to be passed to the recipient then applying the protection is not transparent to the application developer that needs to take into account the transformation done. Then the automatic transformation done by ACTC has little value here since the developer needs to change its code.

The initial idea of this protection was that because debugging a parallel processing is difficult then attacking this kind of code should be more difficult as well and this could be a way to obfuscate the code. This is still a valid approach but as it is implemented now the code is still sensitive to attacks and more theoretical work should be done to provide a more secure protection with the help of mathematics.

Section 6 Task T2.5: Anti-Tampering

Task T2.5 is the only task in WP2 that only started in year 2, and still continues into year 3 of the project. For that reason, no final results are reported yet in this section.

6.1 Anti-Debugging

Section Authors: Bjorn De Sutter, Bart Coppens (UGent)

At UGent, Joris Wijnant (a master thesis student advised by Bert Abrath and other lab members) developed an initial implementation of an anti-debugging extension for Diablo. This implements the anti-debugging architecture described in Section 3.2 of D1.04 v2.0.

During Joris' thesis research, he partially implemented the following parts of the reference architecture:

- Single basic blocks are automatically rewritten by Diablo to be run in the context of a debugger, i.e., in the debugger process, rather than in the context of the original application. This includes injecting code to transfer the registers used and defined by the basic block by means of the ptrace API.
- Diablo inserts instructions that cause a switch to the debugger in the protected application in the places where the rewritten basic blocks should be executed. These instructions also contain meta-information for the debugger component describing the location of the rewritten basic block.
- The debugger component itself. This component attaches itself as a debugger to the protected application, and handles the exceptions triggered by the protected application.
- Diablo injects the debugger component, and ensures that it is started on application initialisation.
- Memory accesses migrated to the debugger as part of the above rewriting are transformed: simply executing them in the debugger would not be correct, as the debugger runs in another memory space. The memory accesses are thus rewritten to work correctly (this will be explained in more detail later on).

While this code was initially written and tested for ARM Linux devices, we already tested the code on rooted Android 4.4 devices, and confirmed that the code works also works in this environment. Furthermore, we have verified with a simple toy app, that the concepts also works on an unrooted Android 5 device. We can hence consider it future-proof at least for the foreseeable future.

Furthermore, we already made the code compatible with the ASPIRE tool chain: it is fully controllable using ASPIRE annotations in the source code.

However, as this code was written by a thesis student, its quality is not yet of a level that is acceptable to be integrated. In particular, it was not yet tested on non-trivial code fragments, such as those that the anti-debugging protection will be used for in the ASPIRE use cases. Applying the technique to those code fragments shows multiple bugs in the transformed code and incorrect, hidden assumptions in the implementation, which we are in the process of fixing at the time of writing (second half M24, October 2014).

We expect to deliver this functionality as part of D2.07 at the end of M24 or slightly thereafter in case unexpected problems still show up. At that time, it will be integrated immediately in the ACTC. We will definitely report on this integration for the third project review.

The memory accesses that are migrated to the debugger context can be transformed in two ways, depending on address accessed. The first way is to replace the memory access by an

invocation of a helper a function that simply invokes the appropriate function from the ptrace API to read/write 4 bytes from the debuggee. This method can be used for any memory access. The second method is more of an optimization to be used when dealing with accesses to the stack: we will use the ptrace API to copy an entire stack region from debuggee to debugger context and point the stack pointer to this copied region when executing in debuggee context. The actual memory accesses will use the changed stack pointer and won't be transformed themselves. When the execution in debugger context is finished will copy the - possibly modified – stack region back from debugger to debuggee context. A comparison between the two methods for a variable number of memory accesses to the stack can be seen in Figure 21, where the stack copy is implemented by copying two memory pages. As one would expect copying an entire region takes about constant in time while the first method of doing a ptrace call for every memory access is linear with the number of accesses. For this specific case we concluded that when doing more as 8 memory accesses to the stack it is preferable to copy the entire region, when doing less as 8 accesses it is better to do them separately, and when doing exactly 8 accesses there is no significant difference between the two methods.

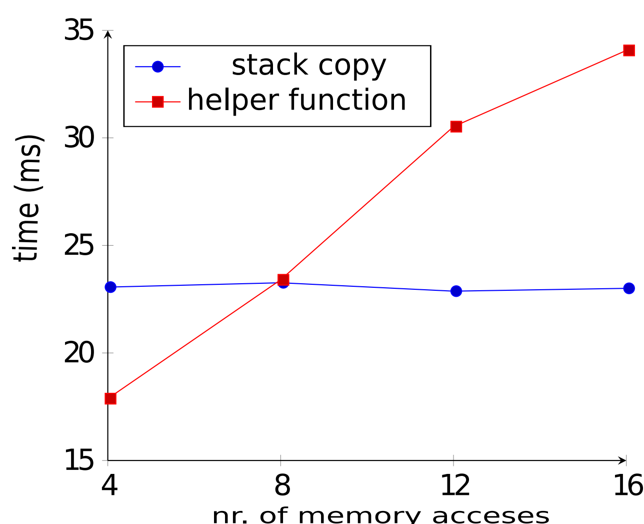

Figure 21 – Comparison of the two methods to transform memory accesses.

The second method could be expanded to be used when transforming a series of memory accesses that use a certain register or constant address as base (as is common when iterating over arrays), but this requires us to know how large the array we will iterate over is in order to copy it, which isn't always known. We will look into how to solve this problem.

6.2 Anti-callback Stack Checks

Section Authors: Bjorn De Sutter, Bart Coppens (UGent)

While not explicitly mentioned in the project DoW, at the time the original DoW was conceived the consortium intended to include heavy-weight anti-callback stacks checks into the set of ASPIRE protections. Those checks would analyze whole stack traces to check that functions in the protected application were not called through unallowed, attacker-injected callbacks from external libraries.

This approach was deemed feasible because, given a whole application to be protected, all libraries that could possibly be loaded at run-time are known at compile time. Furthermore, besides the application binary's entry point, most applications would feature very few

functions intended to be called from external libraries through call-backs, so all allowed call-backs from the libraries can easily be specified. In the case of protected application binary, it is the binary that starts executing first, and that in a sense controls which library routines are invoked.

However, early on in the project, in the initial phase of WP1, the original choice for protecting applications was revised. To enable validation and demonstration of the developed tools and protections on more realistic use cases, it was decided that we would protect dynamically linked libraries. With respect to anti-callback stack checks, this completely changes the scenarios that need to be handled. In a dynamically linked library, by definition a range of functions is exposed to be invoked by external libraries and by the application for which the libraries are loaded. Moreover, in the case of Android, that application and the external libraries are not a simple application. In most cases, and in two of the project's use cases, they are Dalvik/ART, Android's run-time environments in which Java applications are executed that invoke the native (protected) libraries through complex Java-to-native interfaces that are supported in the run-time environments.

Developing code that can walk and analyze complete stack traces in such a context, and reliably decide whether those traces conform to normal execution or instead imply an ongoing attack might still be possible, but it is certainly not possible within the resources foreseen for this task in the project.

The initial high aim of this task has been revised, and somewhat lowered. The proposed implementation of the anti-callback checks is still in line with the DoW, but is less advanced.

Concretely, UGent implemented the necessary functionality in Diablo to inject small stack checks into a dynamic library at the entry points of functions that should not be invoked from outside the library. These checks inspect the return address of the last call and check whether or not it comes from inside the code segment of the protected library itself, or from the outside. In the latter case, a reaction will be triggered. This can be an immediate reaction such as a crash or abort (in case the execution of the function should be blocked immediately for security reasons) or a delayed reaction (in case it is okay for the program to continue executing for a short while).

We already have an embryonic implementation of these call stack checks, that is part of the prototype deliverable D2.07 of M24. However, this implementation is as yet too rough to be integrated in the ASPIRE tool chain in WP5. We foresee to be able to integrate this by the end of M25, and will report on this in the third project review.

6.3 Control Flow Tagging

Section Authors: Jerome D'Annoville (GTO)

The Control Flow Tagging protection aims to check that some assertions are verified during the execution of the application. Gates are added to the code of the application. Each time the activation of the application enters a Gate then the associated counter is incremented; this is the tagging step. The assertions to be verified combine the values of these counters in logical expressions. These assertions are extra controls that are added also to the application. At certain nodes in the graph of the application the assertions verify that the activation has entered the expected Gates. If one or several Gates have been missed then the reaction logic is triggered.

The verification of the assertions can be done either locally or remotely. Advantage of a remote processing is that attacker has no access to the content of the assertions and cannot predict or influence the verdict done by the verifier. The main drawback is that the reaction component on the client side is easy to find and to be blocked. There is no satisfying clue today to embed this component more tightly with the application in order to prevent its detection by the attacker. Indeed, the Reaction Waiting Unit as described in the section 5 of [D3.04] runs in a separate thread due to the constraint of the communication protocol and as

a consequence is loosely coupled with the application. Another constraint is that in case the application does not need the network and might be offline by nature after its installation and setting step then bringing the connectivity constraint only for enabling a possible reaction action is a very strong constraint.

Unlike the online design, keeping the verification of the assertions within the application enables a tight coupling. The issue to solve is to hide the various verifiers part in the code to prevent the attacker to make an easy connection with the reaction part.

The attestor part of this protection, the Gates, already has this constraint to be hidden at the maximum to make it difficult to spot. An early prototype was using big integers to enable to map several Gate counters on a single variable. Each Gate is associated with a different prime number. An Accumulator variable is grouping several Gate counters. It is managed as a big integer except that this data type is not available in C. The Accumulator is initialized to 1 and each time a Gate is entered then the Accumulator is multiplied by the prime number associated to the Gate. Verifiers will be able to retrieve the Gate counter value by dividing the Accumulator.

The extra code could be added at source level. Advantage is that it is easier to implement and the source code inserted in the application would be protected by many other protections of ASPIRE. Still, the choice is to insert the code at binary level to hide at the maximum the extra code. Immediate values used to access the Gate counters are artificially made different in the Gate part and in the assertions part.

An intermediate approach would be to have a combined approach by inserting code at source and binary level: the Gates would be inserted at source level and the assertions processing at binary level. This is possible but not considered today mainly because Gates code is considered as useless code by the optimizer and dropped during the compilation step. This can be mitigated by inserting a call to a dummy external function with the counter values as arguments at the place where assertions have to be verified. Then the binary transformation would be to replace this artificial call to an external function by the assertion code.

As already mentioned in the reference architecture [D1.04], the code integrity checking provide by the Code Guards would prevent the attacker to tamper with the extra code added by the Control Flow Tagging protection. Still, the Gate counter variables have to be protected and a checksum control can be added. It is not considered today because it would make the code referencing the counters data bigger and may attract the focus of the attacker. The option taken in the balance between security and light code is to prefer a discreet protection.

The Control Flow Tagging will be implemented using the Diablo framework.

6.4 Code Guards

Section Authors: Bjorn De Sutter, Bart Coppens (UGent)

For code guards, UGent looked into how to best re-use (where possible) the components that have already been implemented for Remote Attestation (RA). Local code guards need the following elements to function: hash functions, hash check functions, and a tamper response.

We will re-use the functionality related to computing the hashes of regions that has been implemented for RA. This functionality currently consists of a set of hashing functions, an Area Data Structure (ADS) that defines the areas to attest, and code that performs a random walk over an area based on this ADS. To prevent replay attacks and to introduce some diversity, this random walk for RA is seeded by the protection server: subsequent walks will attest different, randomized subsets of the protected area. This also means that the computed hashes will vary over time. In the RA scenario, this is not an issue: the server has all the information to verify the correctness of the hash produced for each random walk.

However, as the protected application needs to verify its own hashes for code guards, and we do want to attest the entire code ranges rather than a subset, this means that we cannot reuse these random walks as-is. Thus, we will ensure that for code guards we deterministically attest the entire protected code region. However, from the point of view of the existing functionality for RA, this can viewed as a 'random walk' that iterates over the entire code range. Thus, nothing needs to fundamentally change for computing the hashes.

As the invocations for the hashing in RA occur asynchronously based on input from the protection server, we cannot re-use the attestation invocation code from RA. However, the locations where hashes need to be computed have been annotated, we will inject the calls to compute the hashes in the annotated locations.

The code to compute the correct hash values can be re-used as-is from the RA implementation. This code runs on the final binary and its ADS, performs the random walk based on the information in the ADS, and finally produces the correct hash value. However, while for RA this information is then stored in a database for later use, we need to inject the correct value in the binary in the correct location. While some effort for this will be required, we anticipate no immediate issues here.

As for RA the verification of the hashes occurs on the server rather than in the protected application, we will need to write and inject custom code that verifies the computed hashes. For each of the hash functions, we will write a simple verification routine in C that compares the correct hash value with the computed value. Calls to these verification routines will be injected in the binary by Diablo on the locations that have been annotated.

We will insert the tamper response in the same manner for other offline attestation techniques, such as call stack checks.

As this proposed implementation for code guards consists mostly of components that have already been integrated, we will only need to integrate the additional step of injecting the correct hash check values into the final binary. As by now the process of integrating additional steps into the ACTC has been streamlined, we anticipate no real problems here.

Section 7 List of Abbreviations

ACCL	ASPIRE Common Client Logic
ACTC	ASPIRE Compiler Tool Chain
ACSL	ASPIRE Common Server Logic
ACTC	ASPIRE Compiler Tool Chain
ADS	Area Data Structure
ADSS	ASPIRE Decision Support System
AES	Advanced Encryption Standard
API	Application Programmer's Interface
ART	Android Run Time
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
DES	Data Encryption Standard
DoW	Description of Work
GUI	Graphical User Interface
IP	Intellectual Property
PRNG	Pseudo Random Number Generator
RA	Remote Attestation
RNC	Residue Number
RTD	Research and Technology Development
SB	(ASPIRE) Steering Board
SVN	Subversion
QAP	Quality Assurance Plan
URL	Uniform Resource Locator
VM	Virtual Machine
WBC	White-Box Cryptography
WBTA	White-Box Tool for ASPIRE
XML	Extended Markup Language

Bibliography

[Abi11] Abin Abraham Alichan, https://www.youtube.com/watch?v=mlzxpkdXP58, May 2011

[D1.04] ASPIRE Project, *D1.04 Reference Architecture*

[D2.01] ASPIRE Project, *D2.01 Early White-Box Cryptography and Data Obfuscation Report*

[D3.04] ASPIRE Project, *D3.04 Intermediate Online Protections Report*

[D5.02] ASPIRE Project, D5.02 *Framework Architecture, Tool Flow, and APIs of the ASPIRE Compiler Tool Chain and Decision Support System*

[Cad08] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. OSDI. Vol. 8. 2008.

[Cle15] Thomas Van Cleemput. Automatic injection of flexible opaque predicates. Master thesis, Ghent University, 2015

[Chr11] M. Chroni and S. D. Nikolopoulos. *Efficient encoding of watermark numbers as reducible permutation graphs*. arXiv preprint arXiv:1110.1194, 2011.

[Col09] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[Col99] C. Collberg and C. Thomborson. *Software watermarking: Models and dynamic embeddings*. In Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 311–324. ACM, 1999.

[Col98] C. Collberg, C. Thomborson, and D. Low. *Manufacturing cheap, resilient, and stealthy opaque constructs*. In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[Cos15] R. Costa and al. *Hiding cryptographic keys of embedded systems*. In Proceedings of the 9th International Conference on Computer Engineering and Applications (CEA '15), 2015.

[Gar79] Garey, Michael R., and David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. 1979. San Francisco, LA: Freeman (1979).

[He02] Y. He and M. Sc. *Tamperproofing a software watermark by encoding constants*. PhD thesis, Computer Science)–University of Auckland, 2002.

[Kal98] B. Kaliski,PKCS #7: Cryptographic Message Syntax, RFC 2315, https://tools.ietf.org/html/rfc2315, March 1998

[Kal00] B. Kaliski, PKCS #5: Password-Based Cryptography Specification Version 2.0, https://tools.ietf.org/html/rfc2898,, Sept. 2000.

[Kar96] Karp, Richard M. *Reducibility among combinatorial problems*. Springer US, 1972. Selman, Bart, David G. Mitchell, and Hector J. Levesque. "Generating hard satisfiability problems." *Artificial intelligence* 81.1 (1996): 17-29.

[Knu69] Knuth, Donald E. (1969). *Seminumerical algorithms. The Art of Computer Programming 2*. Reading, MA: Addison–Wesley. pp. 139–140.

[Lat04] Lattner, Chris, and Vikram Adve. *LLVM: A compilation framework for lifelong program analysis & transformation*. Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004.

[Mos07] A. Moser, C. Kruegel, and E. Kirda. *Limits of static analysis for malware detection.* In Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual, pages 421–430. IEEE, 2007.

[Mut11] Mutsuo Saito, Makoto Matsumoto, Tiny Mersenne Twister (tinymt). http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html, 2011

[Pal00] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. *Experience with software watermarking*. In Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference, pages 308–316. IEEE, 2000.

[Ram94] G. Ramalingam. *The undecidability of aliasing*. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(5):1467–1471, 1994.

[Sel96] Selman, Bart, David G. Mitchell, and Hector J. Levesque. *Generating hard satisfiability problems.* Artificial intelligence 81.1 (1996): 17-29.

[Wan11] Z. Wang, J. Ming, C. Jia, and D. Gao. *Linear obfuscation to combat symbolic execution*. In Computer Security–ESORICS 2011, pages 210–226. Springer, 2011.

[Yad15] Yadegari, Brian Johannesmeyer, Benjamin Whitely, Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code, with Babak
Proc. 36th IEEE Symposium on Security and Privacy, May 2015.