SEVENTH FRAMEWORK PROGRAMME

Advanced Software Protection:
Integration, Research and Exploitation

D5.11
ASPIRE Framework Report

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	November 1, 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10
Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D5.11
WP and tasks contributing:	WP 5 / Task 5.1, Task 5.2
Due date:	October 2016 – M36
Actual submission date:	30 November 2016
Responsible Organization:	POLITO
Editor:	Cataldo Basile
Dissemination level:	Public
Revision:	v1.0

Abstract:
This deliverable presents the final versions at M36 of the ASPIRE Tool Chain and of the ASPIRE Decision Support System.
Keywords:
ASPIRE ASPIRE Tool Chain, ASPIRE Decision Support System

Editor
Cataldo Basile (POLITO)

Contributors (ordered according to beneficiary numbers)
Bart Coppens, Jeroen Van Cleemput, Bjorn De Sutter, Bart Abrath, Jens Van den Broeck, Jonas Maebe, Sander Bogaert (UGent)
Cataldo Basile, Daniele Canavese, Leonardo Regano, Marco Torchiano (POLITO)
Brecht Wyseur, Patrick Hachemane (NAGRA)
Mariano Ceccato, Roberto Tiella, Andrea Avancini (FBK)
Alessandro Cabutto, Paolo Falcarin (UEL)
Werner Dondl, Andreas Weber (SFNT)
Jerome d'Annoville, Paul Gunawan Hariyanto (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 no. 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 report presents the final results of WP5, in which all the protections and software evaluation methodology results obtained in WPs 2, 3, and 4. are integrated. This report also serves as documentation of the deliverable D5.10, which is the final prototype deliverable of WP5 that includes last prototype implementations of the ASPIRE Compiler Tool Chain (ACTC) and ASPIRE Decision Support System (ADSS).

There are three major contributions in this deliverable: (1) the specification of the final ASPIRE annotations, (2) the presentation of the final ACTC, and (3) the presentation of the (ADSS).

The annotations rely on attributes and Pragmas, which are code annotation mechanisms supported by most compilers. The protection specific annotations drive the ACTC when applying protections on data and code. The security requirement annotations are used to indicate the assets, i.e., the data and areas of the source code, and to associate them to the security properties that must be ensured on them. All the annotations are specified semi-formally with a general semantics which is described and illustrated on examples in the Appendices for all the ASPIRE protections.

The ACTC consist of several parts: the source-level tools, a standard compiler, the binary-level tools, the server-side deployment tools, and the metrics tools. To visualize their components and flow uniformly, a template has been designed and used to describe the ACTC parts. The source-level tool flow design consists of 12 ordered passes that implement all source-level protections (some of which co-exist with binary-level counterparts), plus preprocessing and annotation extraction. For compatibility with Diablo, the central tool in the binary-level rewriting part of the ACTC, a number of patches for the standard compiler, assembler and linker used in the ACTC have been developed. The binary-level rewriting part operates in four steps: (1) code analysis and native code extraction, (2) bytecode and virtual machine code generation, (3) integration of the bytecode and the VM code, together with other pre-compiled protection libraries, and (4) and binary-level transformations to implement a range of binary-level protections. The ACTC and the binary-level processing tools have been extended to enable the automated generation of complexity metrics in support of the protection strength evaluation. Furthermore, the ACTC has been extended with a number of scripts to automatically prepare, set-up and run the server-side logic and data of the online protections deployed in a built application. To improve the performance, a caching system is also used. Starting from the analysis of the annotations, the caching system stores the results of all the application phases in such a way that if only the n-th step (protection) the ACTC has to apply is changed, it is possible to reuse all the previous $n-1$ step results. The virtual machine that contains all the ACTC and all the developed protection tools, which was created since Y1, has been regularly maintained to reproduce results easily and to collaborate on the integration of the individual protection techniques of the different partners. With a simple, "Hello, World!"-like program, the deployment of the whole ACTC is demonstrated.

The ADSS is the tool that helps software developers and SW protection experts when they have to protect applications. Starting from the annotated source code, the ADSS generates the golden combinations, i.e., the set of protections that best mitigate the risks against the application assets, and drives, by means of protection specific annotations, the ACTC to apply the golden combination. The ADSS also outputs logs and reports that explain the entire decision process.

The golden combination is selected with a complex work-flow. Initially, all the information about the application to protect is gathered using compiler tools from the source code. Then, the threats against the application assets are determined via a backward reasoning system that identifies all the attack paths. Next, the possible mitigations, i.e., the software protections that protect against the assets, are identified and combined to find the best mitigation. The golden combination is selected by means of a game theoretic approach based on customized minimax search tree and several pruning and reduction techniques (alpha-beta pruning with aspiration windows, iterative deepening with transposition tables, razoring, futility margin, extended futility margin and reductions based on the node scores). Evaluation of protections effectiveness (in isolation and in combination) is based on experts judgments captured in the knowledge base, and estimated with

the models developed in WP4. Finally, an optional ILP optimization model finds the best way to hide the protected assets and delay attacks. Assets are hidden by protecting with same protection techniques in the golden combination other pasts of the application or by extending the protections proposed in the golden combinations to areas outside the assets, until the user defined overhead constraints are saturated.

Contents

1 Introduction 1

I Source-code annotations 3

2 Annotation Basics 3
2.1 Source Code Annotation Mechanisms 4
2.1.1 GCC Attributes 4
2.1.2 Pragmas 4
2.2 Basic Concepts, Infrastructure and Notation 5
2.3 Protection Annotations 5
2.3.1 Data Annotations 6
2.3.2 Code Annotations 6
2.3.3 General semantics 6
2.3.4 Protection-Specific Syntax 7

3 Security Requirements Annotations 8

II The ASPIRE Compiler Tool Chain 9

4 Template for ASPIRE Compiler Tool Chain 9

5 Configuration of the ACTC 11

6 The Source-level ASPIRE Compiler Tool Chain 12
6.1 Ordering the Source-Level Protections 14
6.2 SLP01: Source Code Annotation and ADSS integration 15
6.3 SLP03: White-Box Cryptography Protection 19
6.4 SLP02: Preprocessing 22
6.5 SLP05: Data Obfuscation Transformations 24
6.6 SLP06: Client-Server Code Splitting Transformations 25
6.7 SLP08: Offline Code Guards Transformations 27
6.8 SLP09: Anti-Cloning Transformations 29
6.9 SLP10: Reaction Unit Transformations 29
6.10 SLP11: Diversified Crypto Library Transformations 30
6.11 SLP12: Control Flow Tagging Transformations 31
6.12 SLP04: Annotation Extraction 33
6.13 SLP07: Remote Attestation Transformations 34

7 Compiler, Assembler and Linker 38
7.1 Compiler Requirements 38
7.2 Compilation and Linking Tool Chain 39

8 Binary Rewriting Tool Chain 41
8.1 Overall Binary Rewriting Approach 41
8.2 Diablo 41
8.2.1 Basic Diablo Operation 41
8.2.2 ASPIRE-specific Diablo Development 44
8.3 Client-Side Code Splitting (SoftVM) 45
8.3.1 BLP01: Native Code Extraction 45
8.3.2 BLP02: Bytecode Generation 46

8.4 BLP03: Code Integration 47
8.5 BLP04 - Part 1: VM Invocation & Relocation Fix-ups 47
8.6 BLP04 - Part 2: Binary-level Protections 48

9 Server-Side Deployment **52**
9.1 Deployment Scripts for Online Protection Techniques 52
9.2 Server side slice 52
9.3 Server side RA components 52
9.4 Renewability Manager 53
9.5 Code Mobility Deployment 53

10 Metrics Generation and Collection **54**

11 Caching ACTC **59**

12 License Tool Example **61**
12.1 License Example Description 61
12.2 Source Code Annotations 61
12.3 ACTC Usage 63
12.4 ACTC Configuration JSON File 64
12.5 Setting the Correct Tool Versions 69
12.6 Compiling the License Example 69
12.7 Graphical Representation of the ACTC Compilation Process 77
12.8 Result of Source Code Transformations 79

13 The ASPIRE Shared Build Environment **81**

III The ASPIRE Decision Support System **82**

14 The ADSS work-flow and research issues towards the golden combinations **82**
14.1 Source code analysis 82
14.1.1 Static code analysis 83
14.1.2 Annotation extraction 84
14.1.3 Execution of user-defined application-specific rules 84
14.1.4 Vanilla application build 84
14.2 Attack paths detection 84
14.2.1 Identification of the protection objectives 84
14.2.2 Attack paths computation 84
14.2.3 Attack steps classification 85
14.3 Protection detection 85
14.4 First level protections discovery 86
14.4.1 Solution walker 87
14.4.2 Solution solver 88
14.5 Second level protections discovery 90
14.6 Solution deployment 92

15 The ADSS tool **93**
15.1 The ADSS Architecture 93
15.1.1 Plug-ins 93
15.1.2 Main components 94
15.2 Expanding the AKB and the ADSS 95
15.2.1 Adding new protection instantiations 95
15.2.2 Adding new ontologies 98

15.2.3 Adding new attack steps . . . 98
15.2.4 Adding a new solver for the L2P MILP problem . . . 101
15.3 API . . . 101
15.3.1 eu.aspire_fp7.adss.akb.AKBUtil . . . 101
15.3.2 eu.aspire_fp7.adss.ADSS . . . 101
15.3.3 eu.aspire_fp7.adss.akb.Model . . . 102

16 Conclusions **104**

IV Appendices **105**

A Data-Specific Annotations **105**
A.1 XOR (1-1 encoding) . . . 105
A.2 Merge Scalar Variables . . . 106
A.3 Residue Number Coding . . . 106
A.4 Convert Static to Procedural Data . . . 107
A.5 Multi-threaded Cryptography . . . 108
A.6 Software Time Bombs . . . 108
A.7 Diversified Cryptographic Library . . . 109

B Code-Specific Annotations **112**
B.1 White-box Cryptography . . . 112
B.2 Client-Side Code Splitting by means of SoftVM . . . 114
B.3 Multi-threaded Cryptography . . . 115
B.4 Anti-Debugging . . . 115
B.5 Call-stack Checks . . . 116
B.6 Code Guards . . . 117
B.7 Binary Code Control Flow Obfuscations . . . 118
B.8 Client-Server Code Splitting by means of Barrier Slicing . . . 120
B.9 Code Mobility . . . 121
B.10 Remote Attestation . . . 122
B.11 Control Flow Tagging . . . 125
B.12 Software Time Bombs . . . 127
B.13 Anti-cloning . . . 127

C JSON Format for Diablo - X-translator Interface **129**

List of Figures

1 Different types of components in the ACTC tool flow. 10
2 Color codes that indicate the responsible for the concerned components. 10
3 Four stage flow chart of the ACTC. 12
4 High-level flow chart of the source-level stage of ACTC. 13
5 ADSS interactions with the ACTC. 16
6 Example of source code patching. 17
7 Example of JSON file. 18
8 Detailed flow chart of source code annotation tool flow in the ACTC 18
9 Detailed flow chart of the white-box cryptography tool flow in the ACTC 20
10 Detailed flow chart of the source-level preprocessing in the ACTC 23
11 Detailed flow chart of the data hiding components in the ACTC 24
12 Detailed flow chart of the client-server splitting tool flow in the ACTC 26
13 Detailed flow chart of the offline code guards tool flow in the ACTC 28
14 Detailed flow chart of the anti-cloning tool flow in the ACTC 29
15 Detailed flow chart of the reaction-unit tool flow in the ACTC 30
16 Detailed flow chart of the diversified crypto library tool flow in the ACTC 31
17 Detailed flow chart of the control flow tagging (CFT) tool flow in the ACTC 32
18 Detailed flow chart of the annotation extraction tool flow in the ACTC 33
19 Detailed flow chart of the remote attestation tool flow in the ACTC 37
20 Compiler and linker part of the ACTC. 40
21 Four steps of the binary-level part of the ACTC 42
22 Inputs and outputs of Diablo. 43
23 Tool flow components for chunk extraction and bytecode generation 45
24 Tool flow components for bytecode generation 47
25 Integration of the SoftVM and application of binary-level protections. 48
26 ACTC metrics subsystem supporting online protection techniques and dynamic metrics of obfuscated binaries 55
27 Fixed input and output folders in the non-caching ACTC 59
28 Caching and annotation rewriting flow 60
29 Graphical representation of the binary part of the ACTC compilation process. 77
30 Graphical representation of the source part of the ACTC compilation process. 78
31 Work-flow of the ADSS. 83
32 Example graph for proving $(region_1, integrity)$. 86
33 Simplified flow-chart of the solution walker algorithm. 87
34 Minimax tree of the ADSS. 88
35 Example of the L2P replication technique. 90
36 Example of the L2P enlargement technique. 91
37 Example of the L2P call graph extension technique. 92
38 ADSS plug-in dependencies. 94
39 Architecture of the ADSS. 95

List of Tables

1 Patches to Clang - LLVM . . . 38
2 Patches to GCC . . . 39
3 Patches to binutils . . . 39
4 Metrics of the ADSS. . . . 93

1 Introduction

Section authors:
Bjorn De Sutter (UGent), Cataldo Basile (POLITO)

This report presents the final results of WP5, in which all the protections and software evaluation methodology results obtained in WPs 2, 3, and 4 are integrated into the ASPIRE Compiler Tool Chain (ACTC) on the one hand, and into the ASPIRE Decision Support Systems (ADSS) on the other hand. This report hence also serves as documentation of the deliverable D5.10, which is the final prototype deliverable of WP5, and which consists of the last iterations of the prototype implementations of the ACTC and ADSS.

This document has been written to be self-contained, such that it presents a complete overview of the current state of the prototypes. It can hence also be considered a manual on the ACTC and the ADSS. This manual serves both the project partners, as third-parties that may start working with the open-sourced parts of the ACTC and the ADSS in the near future. For this reason, this document is a mixture of existing content and of new content. Where existing content of older reports is re-used, this is clearly marked at the beginning of sections, together with an indication of the revising and the updating that has been done.

This document is split in 3 major parts.

The first part, which consists of Sections 2 and 3 plus appendices A and B describes the **ASPIRE annotations** that developers can use to mark the assets, their security requirements, and the protections to be deployed on those assets.

The second part discusses the **ACTC** in detail. Over 10 sections and appendix C, all relevant aspects are discussed, including the template use to visualize the tool flow throughout the document, the way the ACTC can be configured, the different source-level processing steps to deploy source-level protection, the compilation and linking of the processed source code, the binary-level protection tool flow, the automated support for generating data and code for the ASPIRE server in support of online protections, the tool support for generating and collecting complexity metrics (in support of the ADSS and the security evaluation methodology), the caching optimizations developed to speed up repetitive recompilation on the ADSS's demand, and the shared build environment that was developed to allow all partners to reproduce each other's results easily.

Furthermore, the deployment of the ACTC on a "Hello, world!"-like example is presented in detail as a beginner's tutorial on the ACTC.

Together with the many online demonstration videos on the ASPIRE-FP7 Demonstration YouTube channel, and the open-source manual D5.13, this second part of this report provides sufficient documentation on the ACTC.

The third part presents the **ADSS**. The ADSS is a framework whose aim is to help software protection experts in automating the process of protecting software applications. To this goal, the ADSS performs a series of analyses to understand the application to protect and to deduce information by means of inferential engines. This information is then used to find the golden combinations, which are the combinations of protections that best mitigate the risks against the application to protect. This part thus presents all the research issues that have been addressed to identify the golden combinations. These issues include (1) the design of a knowledge base, populated by means of analyses of the application source code, which is detailed enough to perform sophisticated reasonings, (2) the development of an inferential engine based on backward programming to discover all the attack paths against the application assets, (3) the realization of a game theoretic approach to determine the golden combinations to protect the assets, and (4) the construction of an optimization ILP model that adds more protections on top of the golden combinations to increase the overall security by rendering in order to attackers more difficult to identify the asset's locations.

Moreover, this part documents the ADSS tool released with the deliverable D5.10. It includes a very detailed description of the tool architecture, together with instructions to install, run, and expand the ADSS.

Part I

Source-code annotations

2 Annotation Basics

Section authors:
Roberto Tiella

ASPIRE aims for providing an ACTC and an ADSS that can be used by non-security experts to protect their applications. Those users are supposed to annotate their software's source code with annotations that describe the assets to be protected, as well as the threats against which protection should be provided. On the basis of those annotations, and aided by the analysis tools, the ADSS automatically determines a suitable configuration to invoke the ACTC components on the code to be protected. For this purpose, we designed a set of annotations that allows the ASPIRE user to annotate his assets and threats.

Even if the ADSS would reach the goal of fully automated, non-assisted protection, user assistance is still compatible with our vision. Some (expert) user assistance or guidance can be still provided, to speedup the solution of the quite complex problem of determining optimal protection automatically. One of the forms in which this expert user assistance can be provided (apart from an interactive ADSS), is to support more concrete source code annotations that guide the ACTC components by instructing those components regarding the protections they have to apply on the annotated code regions, program points or program data. For this reason, we designed a set of annotations that allows the tool chain user to annotate the code fragments, program points, and program data with concrete specifications of protections to be applied. To limit the search spaces for the ADSS to explore, we provided annotations that cannot only prescribe specific protections to be applied, but also a range of options from which the ADSS has to choose.

For example, the above scenarios correspond to the very abstract marking of a procedure in the application as an asset on one extreme side, and the very concrete marking of a procedure as the subject of control flow flattening obfuscation with parameters X and Y and the subject of mobile code obfuscation with parameter W on the other extreme. In between those two extremes, we can also specify simply that a procedure needs to be obfuscated, such that the ADSS uses this annotation and the code analysis results to select the concrete obfuscations to apply to that procedure. So, apart from very concrete protection prescription annotations, we also elaborated more abstract ones. Such more abstract protection prescriptions represent the knowledge of expert user, that is added to the code to restrict the ADSS search space.

Of course, the annotations prescribing precise protections to be applied on code fragments can also be used by users what want to control the operation of the ACTC completely manually, thus side-stepping the ACTC. In fact, during the project those precise protection annotations were the preferred mechanism to perform tests and experiments with the ACTC and its protections under development.

In this part of this deliverable, we first present a brief overview of existing techniques for annotating source code, and discuss the choices made in the ASPIRE project. We then present the basic building blocks of the annotations we have designed for the ASPIRE tool chain. Next, we give an overview of the annotations we designed for the protections foreseen in the project DoW and of which the software architecture was presented in D1.04. This will include very concrete annotations as well as more abstract ones. Finally, we discuss the annotations designed so far for annotating assets and threats. These are in line with the overview on assets as presented in D1.02 Section 3.

2.1 Source Code Annotation Mechanisms

The program transformation process implemented in the ACTC is driven and controlled by inserting ASPIRE annotations in the source code. Developers can annotate source code to specify protection requirements and any application-specific security requirements supported by the ACTC. Source code is annotated by leveraging mechanisms already available in modern compilers: *GCC attributes* and *C99 Pragmas*.

2.1.1 GCC Attributes

As described in Section 6.3 of GNU Compilers Documentation [4], the `__attribute__` keyword allows the developer to specify special attributes when making a declaration. An attribute specifier has the following form:

```
__attribute__ ((attribute-list))
```

An `attribute-list` is a possibly empty comma-separated sequence of attributes, where each attribute is one of the following:

- Empty. Empty attributes are ignored.
- A word (which may be an identifier such as `unused`, or a reserved word such as `const`).
- A word, followed by, in parentheses, parameters for the attribute. These parameters take one of the following forms:
  - An identifier. For example, `mode` attributes use this form.
  - An identifier followed by a comma and a non-empty comma-separated list of expressions. For example, `format` attributes use this form.
  - A possibly empty comma-separated list of expressions. For example, `format_arg` attributes use this form with the list being a single integer constant expression, and `alias` attributes use this form with the list being a single string constant.

The following code fragment illustrates two possible uses of GCC attributes:

```
int x __attribute__ ((aligned (16))) = 0;
struct foo {
  char a;
  int x[2] __attribute__ ((packed));
};
```

While GCC attributes were firstly introduced as GNU-specific extension to the C language, nowadays this feature is also supported by other compilers, e.g., by the LLVM front-end Clang. One possible drawback in using a notation based on GCC attributes is that most compilers by default report a warning for any use of unrecognized attributes. The drawback can be easily circumvented by wrapping the compiler with a script that filters out the specific warning.

2.1.2 Pragmas

As reported in Section 7 of GNU Compilers Documentation [5], the `#pragma` directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. Three forms of this directive (commonly known as Pragmas) are specified by the 1999 C standard. A C compiler is free to attach any meaning it likes to other Pragmas. C99 introduces the `_Pragma` operator. This feature addresses a major problem

with #pragma: being a directive, it cannot be produced as the result of macro expansion. _Pragma is an operator, much like sizeof or defined, and can be embedded in a macro. Its syntax is:

```
_Pragma (string-literal)
```

where string-literal can be either a normal or wide-character string literal. The result is then processed as if it had appeared as the right hand side of a #pragma directive.

2.2 Basic Concepts, Infrastructure and Notation

In the remainder of this document, annotation grammar is specified using the Extended Backus-Naur Form (EBNF). Non-terminals are written in angled brackets as in <INTEGER>. Terminals are written in plain text. Square brackets are used to bracket sequences of terminals/non-terminals. The star symbol '*' specifies zero or more occurrences while the plus symbol '+' denotes one or more occurrences. Finally, ranges of characters are defined using double dots such as in 'A .. Z'.

The following are some general productions used in the rest of the document:

```
<ID> ::= [ A .. Z a .. z _ ][A .. Z a .. z 0 .. 9 _ ]*
         // i.e., C identifiers

<INTEGER> ::= [ 0 .. 9 ]+

<INTEGRAL_SIZE> ::= <INTEGER>

<LIST_OF_INTEGERS> ::= ( <INTEGER> [ , <INTEGER> ]* )

<LIST_OF_IDS> ::=  ( <ID> [ , <ID> ]* )

<SQ_STRING> ::= ' <CHAR>* '

<PEXPR> ::= <INTEGER> | <SQ_STRING> | <ID> [ ( <PEXPR> [ , <PEXPR>]* ) ]
```

Examples: A <SQ_STRING>

```
'this is a sq_string'
```

Some instances of <PEXPR>:

```
12
alpha
key(random(0,10))
algorithm('ROT 13')
```

2.3 Protection Annotations

Protection annotations specify which protections have to be applied to the program. Each protection requires a different set of attributes. Protection annotations are divided in two broad classes: *data annotations* and *code annotations*. There is a also protection called 'none' that specifies that no protections have to be applied to data or code fragments.

2.3.1 Data Annotations

Data are annotated using the GCC feature "__attribute__". The general form is:

```
__attribute__ ((ASPIRE("<DATA_ANNOTATION>+")))
```

<DATA_ANNOTATION> can be either a protection specification or a security requirement. See Sections 3 for a detailed description of security requirements.

```
<DATA_ANNOTATION> ::= <PROTECTION_ANNOT>
                    | <SECURITY_REQUIREMENT>
```

A protection specification is defined by a protection name and some protection-specific parameters:

```
<PROTECTION_ANNOT> ::= protection(<PROTECTION_NAME> [, <PROTECTION_PARAMETER>]*)
```

<PROTECTION_PARAMETER> specifies a particular parameter's value. Its general syntax is:

```
<PROTECTION_PARAMETER> ::= <ID> ( <PEXPR>+ )
```

Appendix A lists the specifications of the protection-specific data annotations we have designed so far.

2.3.2 Code Annotations

Code, i.e., code fragments, are annotated using "_Pragma". There are two constructs:

```
<CODE_ANNOTATION_SCOPE_BEGIN> ::= _Pragma("ASPIRE begin <CODE_ANNOTATION>+")

<CODE_ANNOTATION_SCOPE_END> ::= _Pragma("ASPIRE end")
```

<CODE_ANNOTATION_SCOPE_BEGIN> is used to begin a "protection scope" (see Section 2.3.3). <CODE_ANNOTATION> specifies some protection characteristics of the program up to the end of the scope. <CODE_ANNOTATION_SCOPE_END> is used to end a protection scope. Protection characteristics associated to the scope are discarded and the protection characteristics of the containing scope are restored.

Appendix B lists the specifications of the protection-specific data annotations we have designed so far.

2.3.3 General semantics

This section specifies the general semantics, i.e., independent from the specific transformation, for annotations in relation to C scopes, nesting, loops, continue, break, jumps, function call and return.

Definition A *protection scope* is a portion of the program that has been associated to specific protection attributes through annotations.

Rules for protection scopes:

1. A *default* protection scope is defined for each compilation unit, i.e., a file.

2. A begin/end code annotations define a protection scope which extends from the next statement after the begin annotation up to the last statement before the paired end annotation.

3. A data annotation adds to the current protection scope a protection assignment on the annotate variable from the point of declaration of the annotate object up to the end of the scope of visibility of the variable.

Furthermore, the current implementation will adhere to the following principles:

1. Protection scopes are forced to be nested by syntax as the "end" tag doesn't contain any further information. For this reason, an "end" tag always closes and must close the innermost open scope.

2. Whether or not a protection propagates to called functions is protection-specific. Some protection techniques can define attributes to control if the propagation is performed but no general mechanism is provided.

3. Protection scopes and C scopes cannot be chained, but should instead be properly nested. For example, the following use of code annotation construct is forbidden because the defined protection scope is chained with the for loop's scope:

```
int f(int n) {

    int s = 0;

    _Pragma("ASPIRE begin ...")

    for (i=0; i<n; i++) {
       _Pragma("ASPIRE end") // <-- FORBIDDEN
       s = s + 1;
    }
}
```

2.3.4 Protection-Specific Syntax

This section describes how the source code can be annotated to require the introduction of specific protections such as xor masking, code guards, anti-cloning, etc.

```
<CODE_ANNOTATION> ::= protection(<PROTECTION_NAME> [, <PROTECTION_PARAMETERS>]*)
```

Semantics:

- `protection(<PROTECTION_NAME> [, <PROTECTION_PARAMETERS> ]*)` : protection named `<PROTECTION_NAME>` is applied in the protection scope opened by its `<CODE_ANNOTATION_SCOPE_BEGIN>`.

3 Security Requirements Annotations

Section authors:
Roberto Tiella, Daniele Canavese

Annotations presented in the previous sections, and detailed for many protections in Appendices A and B constitute a mean to specify concretely how a part of the code or a variable is to be protected. Using such annotations might be not affordable for a non-expert developer. Developers may prefer to reason in terms of security requirements: confidentiality, privacy, integrity, etc. of assets exposed in the source code.

Security requirement annotations are devoted to associate an asset with a set of security requirements. The syntax is:

```
<SECURITY_REQUIREMENT> ::= requirement(<SECURITY_ATTRIBUTE>)
```

<SECURITY_ATTRIBUTE> are taken from the table 'Asset categories' in Section 3 "Assets" of the attack model presented in ASPIRE deliverable D1.02 [3] with some changes. First, we removed the support for the non repudiation and execution correctness properties. This last requirement is, however, partially and indirectly supported by specifying the integrity keyword (we decided to reduce the execution correctness' importance since it does not represent a crucial feature in all the ASPIRE use cases). Second, we added the support for a new property: the hard confidentiality requirement. The *confidentiality* property is breached when the attacker can gain the knowledge of an asset at least once during the program execution, while the *hard confidentiality* is broken when this knowledge can be continuously achieved at every moment during the program execution.

The grammar for specifying the security requirements is:

```
<SECURITY_ATTRIBUTE> ::= confidentiality
                       | hardConfidentiality
                       | privacy
                       | integrity
```

Example: The developer annotates an integer variable that requires protection with the requirement 'confidentiality':

```
int x __attribute__((ASPIRE("requirement(confidentiality)")) ;
```

The security annotation, along with all the others in the code, is evaluated by the ADSS and converted into a concrete protection strategy. The ADSS configures the ACTC for the concrete protection and the transformation is applied.

Part II
The ASPIRE Compiler Tool Chain

4 Template for ASPIRE Compiler Tool Chain

Source: Deliverable D5.01b, Section 6, reformatted figures and lightly edited text

Section authors:
Bjorn De Sutter, Jeroen Van Cleemput (UGent)

To enable clear communication within the consortium with respect to the operation of the ACTC and with respect to the responsibilities of the different components and partners, we agreed to use a template for visualizing the tool flow. This template consists of three parts.

First, Figure 1 shows the way in which different types of tool chain components are visualized.

Secondly, Figure 2 shows the different colors that are used in tool flow diagrams to denote the responsible project partners for the concerned components.

Thirdly, naming and numbering conventions have been agreed on for documents, i.e., input files, output files, and intermediate files:

- **SCxx** Source code document or directory nr. xx, where the following file name extensions are foreseen:
  - **.c**: C source code
  - **.h**: C source code header
  - **.cpp**: C++ source code
  - **.hpp**: C++ source code header
  - **.i**: Pre-processed C source code
- **BCxx**: Binary code document or directory nr. xx, i.e., object files, dynamically linked libraries, or executables, where the following file name extensions are foreseen:
  - **.o**: Object file
  - **.a**: Archive of object files to be linked statically
  - **.so**: Dynamically linked library
  - **.out**: Binary executable program
- **SLPxx**: Source-level software processing step nr. xx
- **SLCxx**: Configuration file for SLPxx
- **SLLxx**: Log file produced by SLPxx
- **BLPxx**: Binary-level software processing step nr. xx
- **BLCxx**: Configuration file for BLPxx
- **BLLxx**: Log file produced by BLPxx
- **Dxx**: General data file nr. xx
- **Mxx**: Metrics file nr. xxk

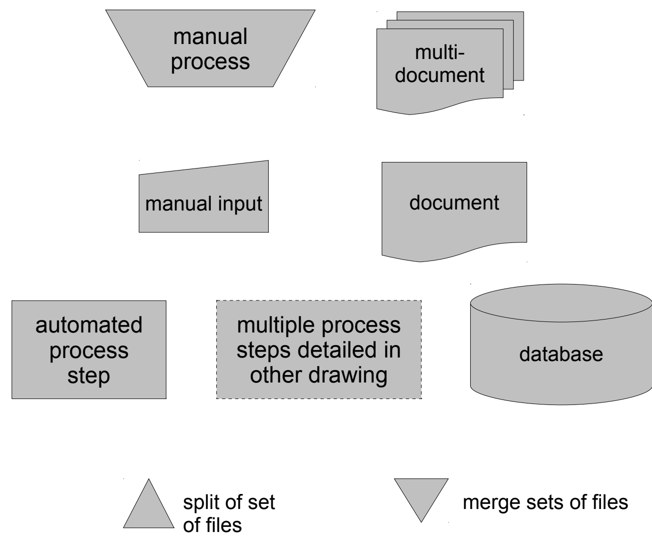

Figure 1: Different types of components in the ACTC tool flow.

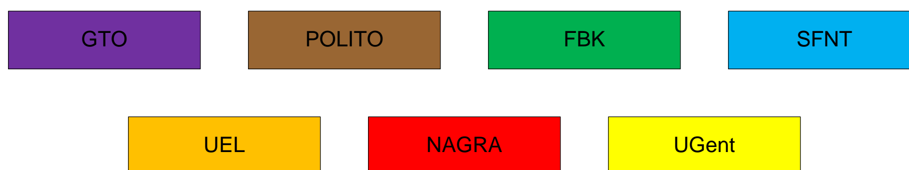

Figure 2: Color codes that indicate the responsible for the concerned components.

5 Configuration of the ACTC

Source: Deliverable 5.06 v1.2 Section 6.4, extended (last paragraph

Configuring the ACTC compilation process is done using an external configuration file. By default this is the `aspire.json` file. The configuration file consists of six sections: `tools`, `src2src`, `src2bin`, `bin2bin`, `server`, and `metrics`.

In the tools section paths to the different tools used by the ACTC can be configured. The `src2src`, `src2bin` and `bin2bin` sections in turn configure the individual compilation phases of the ACTC. Each compilation step can be configured to be excluded and/or traversed. Excluding a compilation step completely skips the step and therefore no output files are generated. Traversing a compilation step copies the source files to the destination folder without any change and is the preferred option to skip a compilation step when subsequent steps depend on the output. Besides the exclude and traverse parameters, each compilation step has an options parameter to provide application specific options to the tool used in that step. Additional compilation step specific parameters have been annotated inline (in red) in the configuration file below.

The `server` section is used the specify server-side deploy scripts for the online protection techniques.

In the `metrics` section it is possible to specify the generated metrics files for each compilation step. These metrics files are then collected during the M01 compilation step and aggregated in the `M01` directory.

Individual configuration options of the different tools are explained in the source-level tool chain Section 6, compiler Section 7 and binary-level tool chain Section 8.

Over the duration of the project, the JSON configuration file has evolved, most notably by adding options and configuration sections for added protections, for server-side processing, and for specifying platform options. By invoking the ACTC with `-u < JSON configuration file >` the ACTC can automatically update a configuration file from an older format to the most recent format.

6 The Source-level ASPIRE Compiler Tool Chain

Source: Deliverable D5.01b Section 7, updated to reflect latest release

Section authors:
Bjorn De Sutter, Jeroen Van Cleemput, Bert Abrath (UGent), Mariano Ceccato, Roberto Tiella, Andrea Avancini (FBK) Patrick Hachemane, Brecht Wyseur (NAGRA), Alessio Viticchié (POLITO), Paul Gunawan Hariyanto (GTO)

The ASPIRE Compiler Tool Chain is not a single tool in practice. It consists of a large set of tools and techniques that operate on different representations of the software to be protected. These tools and techniques can roughly be split in four subsets, corresponding to the four stages depicted in the flow chart in Figure 3.

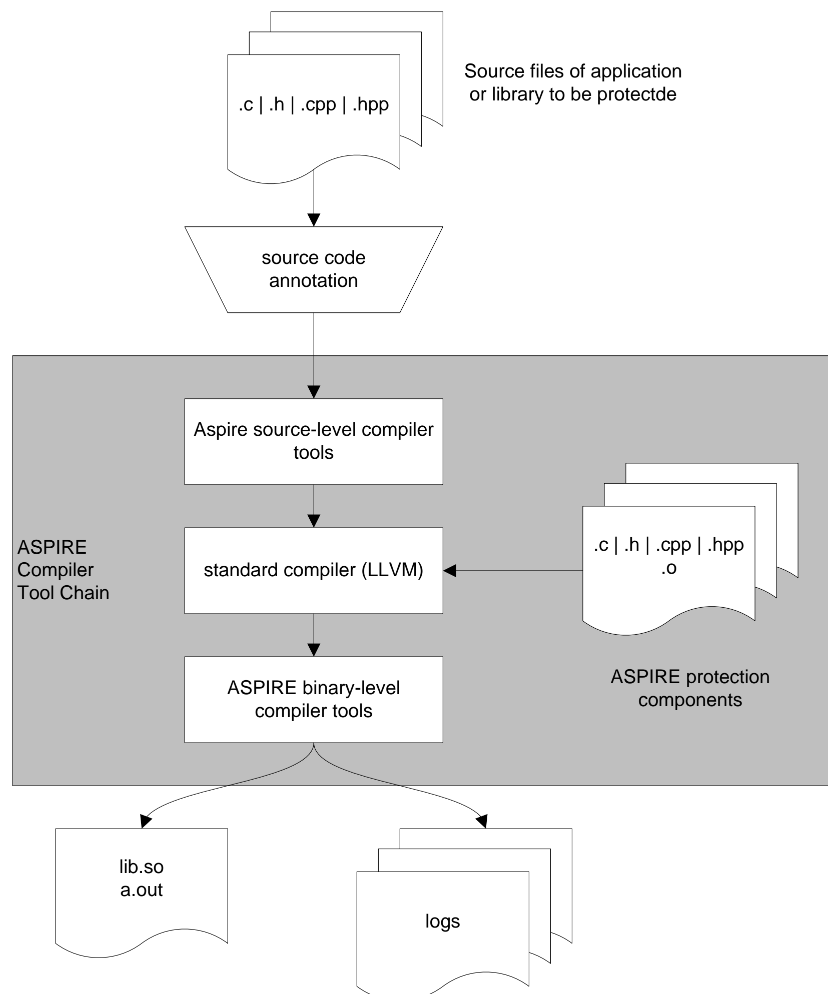

Figure 3: Four stage flow chart of the ACTC.

- Source code annotation is to be performed manually by the user of the tool chain to inform the tool chain about the security requirements and protections to apply. The ADSS can replace security requirement annotations by protection annotations.
- Source-level compiler tools analyze and transform the software source code to implement a number of protections that are best implemented at that level of abstractions.
- A standard-compliant compiler and assembler and linker compile the protected source code into object files and link them, together with any additional protection components that are

not part of the original application, but that need to be linked into the protected application in support of the protections, as described in deliverable D1.04. For the standard-compliant compiler, we have selected LLVM 3.4, and for the standard-compliant assembler and linker, we have selected binutils 2.23.2.

- Binary-level compiler tools, i.e., link-time rewriting tools, rewrite the binary code in the object files to apply additional protections.

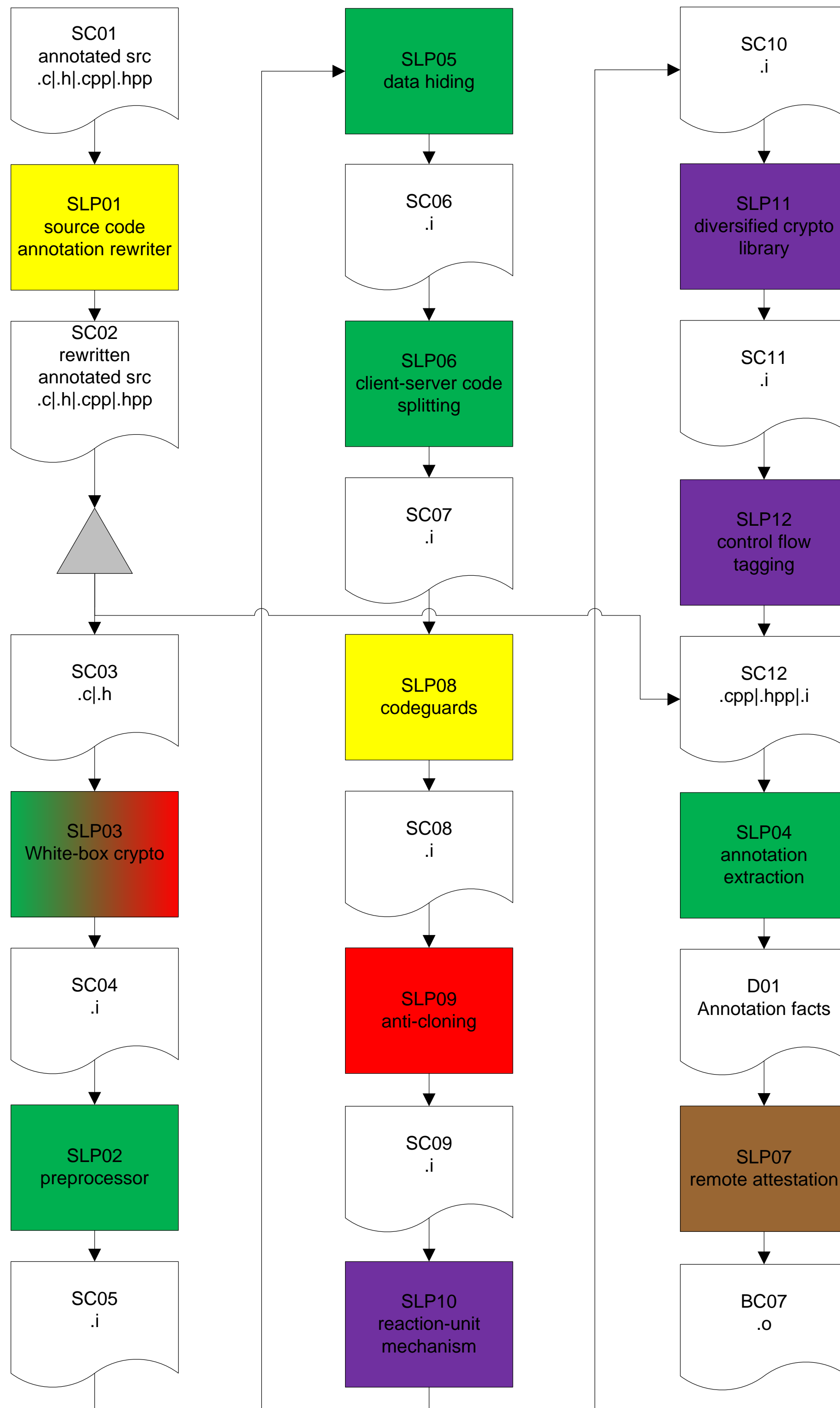

Figure 4: High-level flow chart of the source-level stage of ACTC.

This section presents the design of the source-level components of ACTC as integrated in the final ASPIRE tool chain (Deliverable D5.10) released in M36 of the project. The binary-level part of the tool flow will be presented in Section 8.

Figure 4 depicts the high-level flow of the source-level ACTC. It includes the following source code transformation tools: An automatic source code annotation rewriting tool, a preprocessing step for the C code to be protected and several code transformation steps: white-box cryptography, data hiding, client-server code splitting, code guards, anti-cloning, reaction units, diversified crypto, control flow tagging and remote attestation. Additionally, the annotations are extracted from the source code to allow the information contained therein to be passed to the binary-level tools.

In Figure 4 as well as in later flow charts of the ACTC, we have omitted the log files that all tools produce, not to overload the flow charts. But obviously all passes in the ACTC are supposed to produce a log of the results of their analysis and of the transformations they applied onto the application code. In the current implementation, this is indeed the case.

Throughout this document, it is important to keep in mind that the whole ACTC is designed to be modular, such that it is able to work without all protection compilation steps being activated, and with interchangeable implementations of the different components implementing the individual protections. This modular design was chosen (1) to enable the independent development of the individual techniques during the project, thus mitigating many risks for delays, (2) to support research into the integration of a large number of techniques without running the risk for intellectual property contamination between the many partners in the project, (3) to ease immediate exploitation of the results by the individual partners, without too much dependencies on the other partner's contributions. The downside is of course that at some points the tool chain might seem more complex than needed for pure technical reasons.

6.1 Ordering the Source-Level Protections

Source: Deliverable D5.01b Section 7.3, updated to reflect latest release

After the code has been annotated by the user, it is ready to be protected, i.e., analyzed and transformed. The ASPIRE DoW mentions several protections that are implemented by means of source-to-source transformations. During a number of conference calls with all consortium partners, we decided that the most appropriate order is the following:

(Steps in italic require both source code and binary code transformations)

SLP01 automatic source code annotation rewriting tool, based on the decisions made by the ADSS

SLP03 white-box cryptography protection

SLP02 a preprocessing step for the C code to be protected

SLP05 data hiding

SLP06 client-server code splitting

SLP08 *offline code guards*

SLP09 anti-cloning

SLP10 reaction unit

SLP11 diversified cryptography library (DCL)

SLP12 *control flow tagging*

SLP04 annotation extraction

SLP07 *remote attestation*

The most important reason for picking this order is composability. White-box cryptography essentially comes down to replacing invocations of standard cryptography primitives by invocations of their white-box counterparts. Those WBC counterparts are generated in source code as part of the ACTC being deployed. That source code is then included in the software to be protected, and from then on, the included code should not be distinguished from the original and transformed application code. In other words, all other protection techniques should also be applicable to the code that implements the WBC primitives. Clearly, that requires the inclusion of those primitives in the software before any other analysis or transformation is applied. Furthermore, the WBC tool requires C source code as input, which is why the tool is run before the preprocessing step.

Similarly, variables of which the value encoding has been changed by means of data hiding transformations, should still be potential candidates for client-server code splitting. The computations on those variables are lifted from the program and migrated to a secure server to be executed out of the observable world of the attacker. This migration requires data values to be transferred between the client application and the ASPIRE protection server. By applying code splitting after data hiding, the splitting automatically takes into account the changed data encodings. If the order had been reversed, the global application of data hiding techniques would have been limited at program points where data is exchanged between the client and the server.

Subsequent transformations only need to be applied at the client side of the application. Because there are no inherent dependencies between the steps, they are executed in order of implementation.

In a second-to-last step the annotations are extracted from the source code by the annotation extraction tool. The remote attestation tool, which requires these extracted annotations as an input, is executed last.

As is clear from the execution order listed above, the numbering of the steps is not the order in which they are executed. Instead, the numbering reflects the order in which the steps were added to the design of the ACTC. This numbering order to some extent also reflects the order in which the steps were integrated into the ACTC implementation, but the mapping is not perfect, because some design decisions were taken long before the corresponding steps were integrated.

6.2 SLP01: Source Code Annotation and ADSS integration

Type: Source only

With the annotations described in Part I and several appendices, the user first has to annotate his source code (SC01 in Figure 4) manually. These annotations can be either security requirement annotations (2.3) or protection annotations (3). During SLP01, the security requirement annotations are rewritten to concrete protection annotations based on an external patch and annotations file provided by the ADSS. These protection annotations can then be interpreted by the different protection transformation tools. Existing protection transformations contained in SC01 are preserved and copied over to the source code in SC02.

Given the limited resources of the project and the fact that no reliable source-level tools are available to operate on C and C++ code combined, the ASPIRE Steering Board has decided to implement support for C code annotation and protection only. Applications or libraries to be protected can contain C++ code as well, e.g., to provide a C++ wrapper interface to external code, but that C++ code is not protected.

The reason for this engineering focus is that it avoids having to implement the source-level analyses and transformations on two different program representations and in two different sets of

tools: one supports C and its syntax, another supports C++ and its syntax.

Obviously, the consortium has made sure that this implementation focus does not build on assumptions that would not hold for C++ code. To the contrary, every design and development has be done in full consideration of their portability to C++. Only the porting itself was not be done within the project.

Because the implemented tools do not handle C++ code, the annotated software code base is split in two parts. The C++ part is copied to the output folder of the final source code transformation step (SC12 in Figure 4) and goes directly to the standard compiler. The C part (SC03 in Figure 4) goes to the actual source-level protection tools.

ADSS integration

Source D5.08 Section 8, minor modifications

The ADSS needs to communicate with the ACTC for two reasons. First, to deploy the golden solution, i.e., rewrite security requirement annotations into concrete protection annotations. This interaction typically happens only once at the end of the ADSS work-flow. Second, to compute the complexity metrics on the application. The metrics are used to estimate the strength of a protection combination and, hence, to find the best one. This happens once at the beginning, when the ADSS needs the metrics computed on the vanilla application and a number of times later when the search for the golden combination is running. During this phase, different combinations of protections are temporarily deployed and the resulting code metrics are extracted in order to evaluate their strength. Figure 5 depicts these interactions.

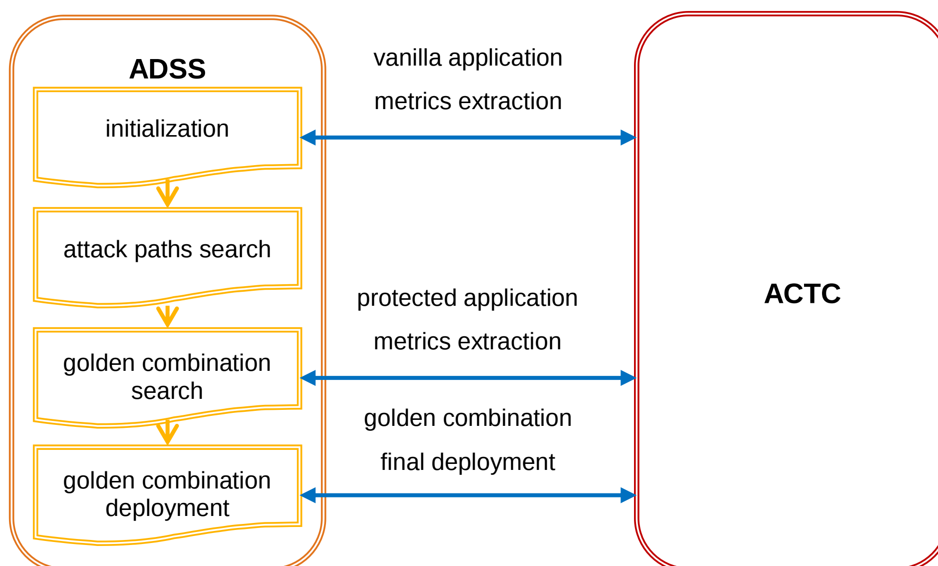

Figure 5: ADSS interactions with the ACTC.

When deploying a combination of protections, both because it is the golden combination or an intermediate combination under analysis, the ADSS is, in a nutshell, rewriting the security requirement annotations into concrete protection annotations. Note that the ADSS might also choose to add some completely new annotations in the code in addition to the original ones. From the ACTC point of view, that means that both annotation rewriting and annotation adding must be supported.

To deploy a combination of protections, the ADSS creates a patch file and a JSON file, and triggers the building of a protected application version by the ACTC. The ACTC then reads these two files, correctly applies the annotations and performs a normal build.

Before performing the compilation phase, the ACTC automatically applies the ADSS patch. The patching is executed on the original C/C++ source files before the pre-processing. This is needed in order to fully support all the protection tools that work on the source files before the pre-processing (e.g., white-box cryptography). The patching is effectively used to introduce a sort of intermediate annotation format, particularly suitable for the ACTC caching. (This caching is described later in more detail in Section 11.) All the security requirement annotations are rewritten and, if needed, several new annotations might also be added. Figure 6 shows a sample code region patching where 2 annotations are rewritten and 1 is added. The new annotations follow the syntax placeholder($< id >$) where $< id >$ is a (zero-based) integer uniquely identifying the annotation amongst the ACTC project.

```
int x __attribute__((ASPIRE("requirement(confidentiality)")));
...
_Pragma("ASPIRE begin requirement(privacy)")
x = (x & 1) << 5;
++y;
_Pragma("ASPIRE end")
for (i = 0; i < z; ++i)
  function(x, y, i);
```

```
int x __attribute__((ASPIRE("placeHolder(0)")));
...
_Pragma("ASPIRE begin placeHolder(1)")
x = (x & 1) << 5;
++y;
_Pragma("ASPIRE end")
_Pragma("ASPIRE begin placeHolder(2)")
for (i = 0; i < z; ++i)
  function(x, y, i);
_Pragma("ASPIRE end")
```

Figure 6: Example of source code patching.

The unique id is automatically computed by the ADSS and it is guaranteed to be unique amongst all the annotations in the same project, regardless of the source files containing them. Note that two annotations in two different projects might have the same id. This value is computed as the index in the list of all the annotations sorted by their source file path name and their order of appearance in the source file.

After the initial patching, the ACTC reads another JSON file and uses its content to replace the placeholder annotations with concrete protection annotations. This file has a straightforward structure and stores the mapping between the unique annotation identifiers and their concrete protection annotations, easily allowing the ACTC to perform the build with its caching facilities. Figure 7 shows an example of such a JSON file.

These files contain an array of JSON objects, each one specifying how the ACTC should translate a specific placeholder annotation into a concrete one. The objects declare the file name containing the annotation to rewrite, its unique id and the annotation content that should replace the placeholder.

ACTC tool chain support

Source D5.08 Section 8, unmodified

To support annotation rewriting, processing step SLP01 was extended from a simple source code fetcher to a source code annotation tool. To this end the tool has been split in three different phases as shown in Figure 8. In a first phase, simply called SLP01, the tool retrieves the source code and copies it to the SC02 directory as usual. A second phase, SLP01_patch, applies the patch file provided by the ADSS to replace the security requirement annotations with the placeholder annotations containing a unique ID. Finally, in a third phase SLP01_annotate the placeholder annotations are replaced with the concrete protection techniques defined in the ADSS provided external anno-

```
[
    {
        "file name": "main.c",
        "id": "0",
        "annotation content": "protection((xor, mask(random))"
    },
    {
        "file name": "main.c",
        "id": "1",
        "annotation content": "protection(anti_debugging(in_debugger))"
    }
]
```

Figure 7: Example of JSON file.

tation JSON file.

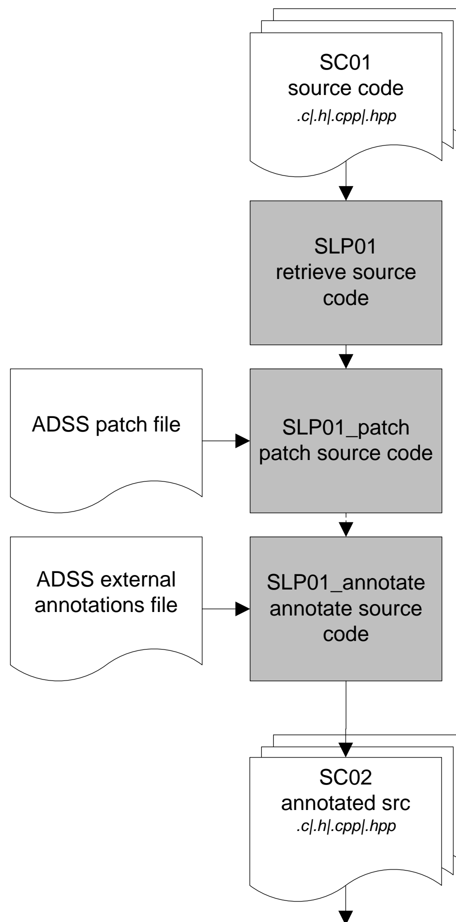

Figure 8: Detailed flow chart of source code annotation tool flow in the ACTC

Configuration

The configuration of the source code annotation rewriting pass is illustrated with the following example, of which the contents speaks for itself:

```
// Source code annotation
"SLP01": {
  "excluded":                false,
  "traverse":                false,
  //Annotations patch file provided by the ADSS
  "annotations_patch":       "adss.patch",
  //External annotations JSON file provided by the ADSS
  "external_annotations":    "adss.json",
  //Application source files
  "source"   :               ["test.c",
                              "test.h",
                              "cpp_test.cpp",
                              "cpp_test.hpp",
                              "../include/*.h"]
},
```

6.3 SLP03: White-Box Cryptography Protection

Source:
Deliverable D5.01b Section 7.4 (updated document folders)
Deliverable D5.08 Section 3.1 (renewability update, unmodified)

Type: Source only

Figure 9 depicts the detailed compilation tool structure of the first protection deployed on source code in the ACTC: white-box cryptography. We refer to deliverables D2.01, D2.04, and D2.08 for a detailed description of the transformations that need to be applied, and to D1.04 v2.1 Section 3.5 for an architectural description of software protected with ASPIRE white-box cryptography techniques.

This flow chart is quite complex because these protection techniques require the rewritten application code to invoke custom C procedures that implement white-box cryptography primitives and that are generated on the fly, albeit in separate files. For such invocations to be valid, the invoked procedures need to be declared in the rewritten source code files, which can most easily be achieved by including (with the `#include` preprocessor directive) the necessary header files. Including header files in already pre-processed code is not a good idea, however, because it risks including the same code multiple times.[1] To avoid this problem, the white-box cryptography tool flow operates as follows.

First, an extraction tool SLP03.01 (co-developed by FBK and NAGRA, based on TXL) extracts the white-box cryptography source-code annotations from the pre-processed source code SC03. These annotations are all protection annotations that prescribe precisely what white-box cryptography protection should be invoked and where. The extracted data is then passed as a configuration file SLC03.02 to SLP03.02, NAGRA's White-Box Tool (WBT), which is described in more detail in D2.04.

This configuration file is an XML file that specifies the requested transformations by means of

[1] Such multiple inclusion is typically prevented through the use of `#define` and `#ifdef` directives, but that only works if all of them are handled in the same preprocessor run, which is no longer the case when code is included in already pre-processed code.

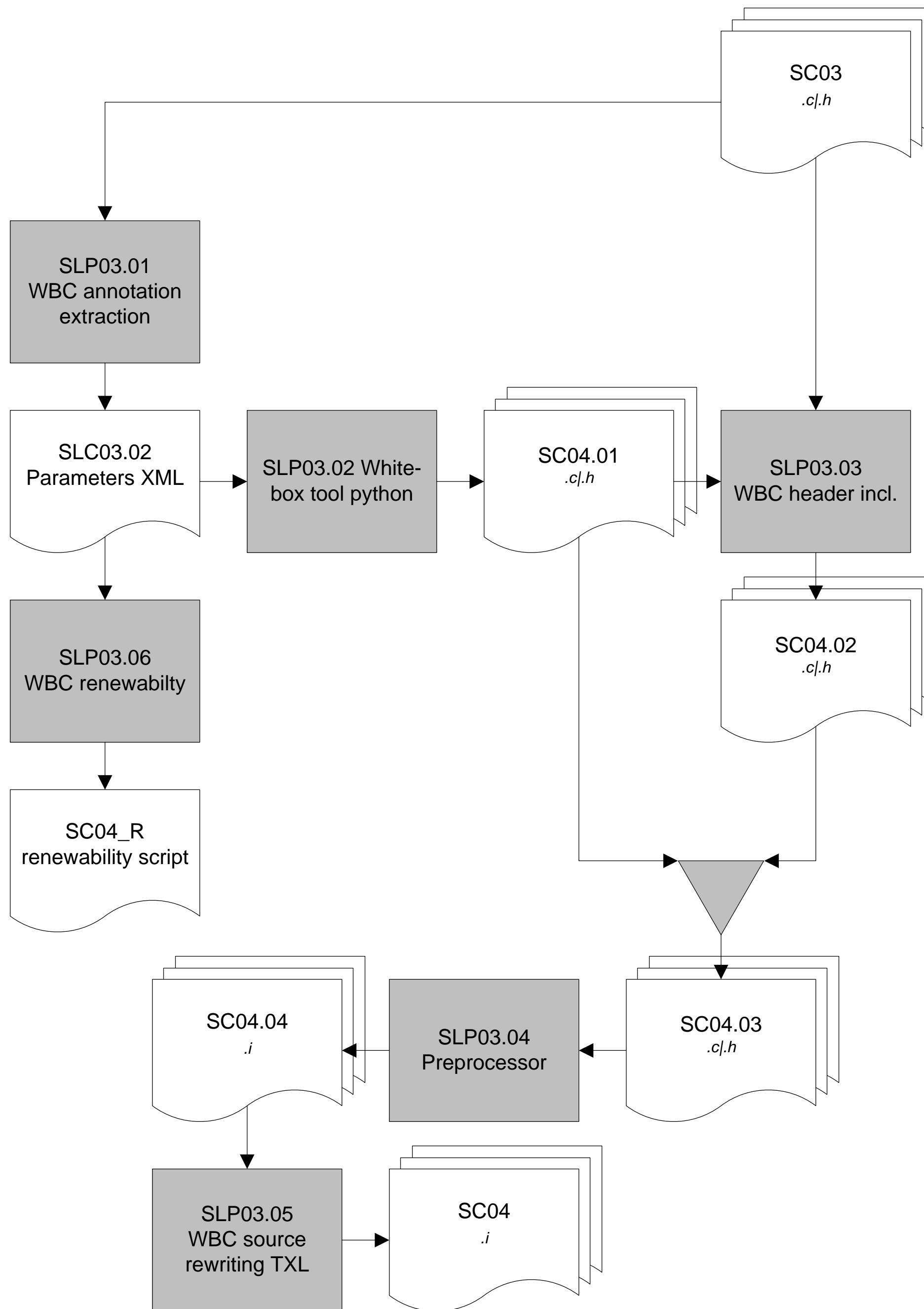

Figure 9: Detailed flow chart of the white-box cryptography tool flow in the ACTC

key-value pairs. This format was chosen because NAGRA's existing tool is already based on XML configuration files.

The keys are:

- *key:* The value of the cipher key. In case the key is dynamic, this value is not specified;
- *key_length:* The length of the cipher key. In case the key is dynamic, this value is not specified;
- *algorithm:* The name of used algorithm (AES, DES, 3DES, ...);
- *operation:* The operation performed by the algorithm (encrypt or decrypt);
- *mode:* The mode of the used algorithm (ECB, CBC, ...);
- *label:* A label to uniquely identify this case; this label is used to build the name of the function (see below)

- *client_file_name:* Name of the file that defines the generated WBC function prototype, without suffix; the header file, with .h suffix, has to be included in the source code.

On the basis of the passed parameters, the WBT generates C code files SC04.01, consisting of .c files in which the required WBC primitives are implemented, as well as of corresponding header files that can be included in the original application. This inclusion, by means of `#include` directives at the top of the .c files of the original application (i.e., part of SC03), is performed in a very simple code rewriting step SLP03.03.

The rewritten .c files and the original program headers then form the code base SC04.02. SC04.01 and SC04.02 are merged into SC04.03, which thus contain all original application code, as well as all white-box crypto primitives. This code is unprocessed C code, however, so it again needs to be pre-processed, hence the additional rewriting step SLP03.04, which re-performs the steps already discussed in Section 6.4.

While the resulting pre-processed code SC04.04 contains all the necessary functionality, the white-box cryptography primitives in it are not invoked yet. To invoke them (instead of their non white-box counterparts in the original code), a final source code rewriting step SLP03.05 is executed, producing the code SC04. In this step, based again on TXL, procedure calls and key values are replaced, as described in detail in deliverables D2.01 and D2.04.

Renewability Support

The WBT tool (more precisely, its implementation for ASPIRE, named WBT for ASPIRE or WBTA, as described in D2.04) can also deliver renewable white-box crypto. For renewable white-box crypto, the tables embedding the (randomized) WBC key are made mobile, and new version of the mobile tables can be generated on a server.

The support for this mechanism uses the existing white-box crypto tool with an additional file as parameter, named decision file; this file contains a random seed that allows renewing the code and data generated by WBTA. Step SLP03.06 takes the extracted annotations in SLC03.02 as input and generates a renewability script in SC04_R for each annotation file. The scripts, when later invoked on a server, re-runs the WBTA tool using a specific decision file containing a new random seed and then pre-processes and compiles the generated source files. The resulting object files are used to update the required mobile data blocks on the server.

Configuration

The configuration of the white-box crypto protection step is illustrated with the following example, in which the comments explain the meaning/goal of the different items:

```
// white-box crypto
"SLP03": {
  "excluded": false,
  "traverse": false,
  // Generate renewability script to regenerate WBTA code and
  // update the required mobile blocks
  "renewability_script": true,
  // WBC seed (random, aid, none)
  "seed": "none",

  // WBC annotation extraction tool
  "_01": {
    "excluded": false
  },

  // White-Box Tool python
  "_02": {
    "excluded": false
  },

  // WBC header inclusion
  "_03": {
    "excluded": false
  },

  // preprocessor
  "_04": {
    "excluded": false
  },

  // WBC source rewriting tool
  "_05": {
    "excluded": false,
    "options":  [""]
  }
},
```

6.4 SLP02: Preprocessing

Source: Deliverable D5.01b Section 7.2, updated to reflect latest release
Type: Source only

In order to apply protections on the annotated source code SC02 to eventually produce protected source code SC06, the annotated source code needs to be read, parsed, analyzed, transformed, and generated again.

The reading, parsing, and generation steps are fairly standard steps. ASPIRE did not aim for pushing the state of the art with respect to these processing steps, so instead we reused as much as possible existing tools. To enable this, the source-level tools need to operate on standard C code that adheres to a fixed grammar that can be handled by existing tools.

Typical C source code does not adhere to such a fixed grammar, however. Instead all kinds of macros and other preprocessor directives are often found in source code. To avoid the huge overhead of handling such (grammatically virtually unrestricted) source code, the ACTC first preprocesses all C code in step SLP02 by means of the C preprocessor that is available from the used C compiler, as shown in Figure 10.

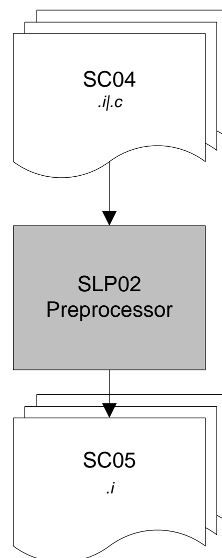

Figure 10: Detailed flow chart of the source-level preprocessing in the ACTC

In the preprocessing step SLP02, one SC05 .i file is generated per SC04 .c file. The .h headers in SC04 are all the header files included in the .c files. This includes application-specific headers, but also, e.g., headers from the standard C libraries. So there is no fixed relation between the number of .h files and the number of .c/.i files.

As stated, this preprocessing allowed the ASPIRE project to focus on the actual code analyses and code transformation steps of the source-level tool chain, i.e., the steps where the protection-specific development tool place and where the ASPIRE consortium wanted to push the state of the art.

Initially SLP02 performed both a preprocessing step and a normalization step (see deliverable D5.01b Section 7.2). The normalization step has since been removed (see deliverable D5.04 Section 2.2.1), leaving only the preprocessing step.

Configuration

```
// preprocessor
"SLP02": {
  "excluded": false
},
```

Preprocessor flags can be specified using the *options* setting of the *PREPROCESS* part of the *src2bin* configuration section:

```
"PREPROCESS": {
  // -I <dir>
  // -isystem <dir>
  // -include <file>
  // -D<macro[=defn]>
  "options"    : ["-I ../path/to/headers",
                  "-I ../path/to/src",
                  "-DVARIABLE"]
},
```

6.5 SLP05: Data Obfuscation Transformations

Source: new content, because tool flow was simplified
Type: Source only

As shown in Figure 11, the data obfuscation processing step is monolithic from the ACTC's perspective. Whereas early implementations of the protection were relying on CodeSurfer to analyze the code in preparation of more global (interprocedural) protections, incl. of aggregate data types such as arrays, the project in the end lacked sufficient resources to extent the data obfuscations in that direction. Instead, local analyses are performed directly on the abstract syntax representation of the source code that TXL builds, and which is updated to deploy the protections as specified by the annotations. A simple bash script SLP05 wraps the necessary invocations of TXL.

The actual data obfuscations that are supported, both static and dynamic instantiations are described in more detail in deliverables D2.01,

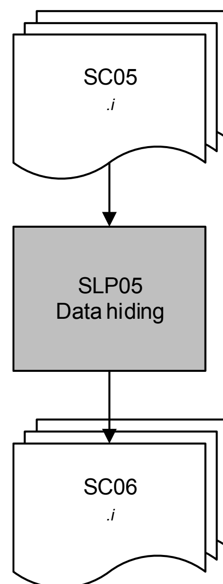

Figure 11: Detailed flow chart of the data hiding components in the ACTC

Whereas inserting invocations to white-box crypto primitives is a relatively and deliverable D2.08.

Configuration

The data hiding plug-in does not require any configuration. As the example configuration below shows,the ACTC still foresees that a preceding data flow analysis of the source code can be invoked as a sub-step SLP05_01, before the actual obfuscation in sub-step SLP05_02. This prepares the ACTC for supporting more global and advanced forms of data obfuscations.

```
// data hiding
"SLP05": {
  "excluded": false,
  "traverse": false,

  // source code analysis
  "_01": {
    "excluded": false,
    "options" : []
  },

  // data obfuscation
  "_02": {
    "excluded": false,
    "options" : []
  }
},
```

6.6 SLP06: Client-Server Code Splitting Transformations

Source: Deliverable D5.06 Section 3.2; Deliverable D5.08 Section 3.3, merged and updated to reflect latest release
Type: Source only

The ACTC integrates client/server code splitting tools as component SLP06 (see Figure 12). The technique applies barrier slicing to identify the portions of the application to move, and a set of transformations to generate the new client application and its trusted server-side code component. The two new components will then execute these portions of code in a synchronous way to preserve the original functionalities of the application (see deliverable D3.04 for further details).

The component (SLP06.01) is responsible for processing the source code of the application to protect. Code in input is analyzed by a TXL program to identify, extract, and manipulate code annotations that refer to client-server code splitting, to produce a set of fact files that contain information related to:

- the portion of code to be subject of the protection;
- critical variables that need to be protected;
- barrier variables, used to stop propagation of dependencies when calculating the barrier slice.

These fact files are stored in a specific folder (indicated as SC06.01/facts in Figure 12), to be later used to drive the computation of the barrier slice. The barrier slice is computed by means of a custom slicing script implemented in CodeSurfer (component SLP06.02). Additional CodeSurfer scripts perform the extraction of another set of fact files in SC06.01/csurf-project, to store information related to uses/definitions of critical and barrier variables, pointers, function calls, formal parameters of functions, data types. Component SLP06.03 generates the protected client application and the server-side code to run the slice, respectively. The component applies on the pre-processed code of the original application, while the facts and the barrier slice extracted with CodeSurfer are used as input for the code transformations. Client and server-side code generations are implemented in TXL. The TXL program applies a set of code transformations to produce the protected client application (SC07) and the server-side sliced code (SCS01). Since M30, SLP06.03 component includes the accl-message-wrapper.c file to handle network communication at client-side. Integration with the ACTC works with the main splitting components described earlier.

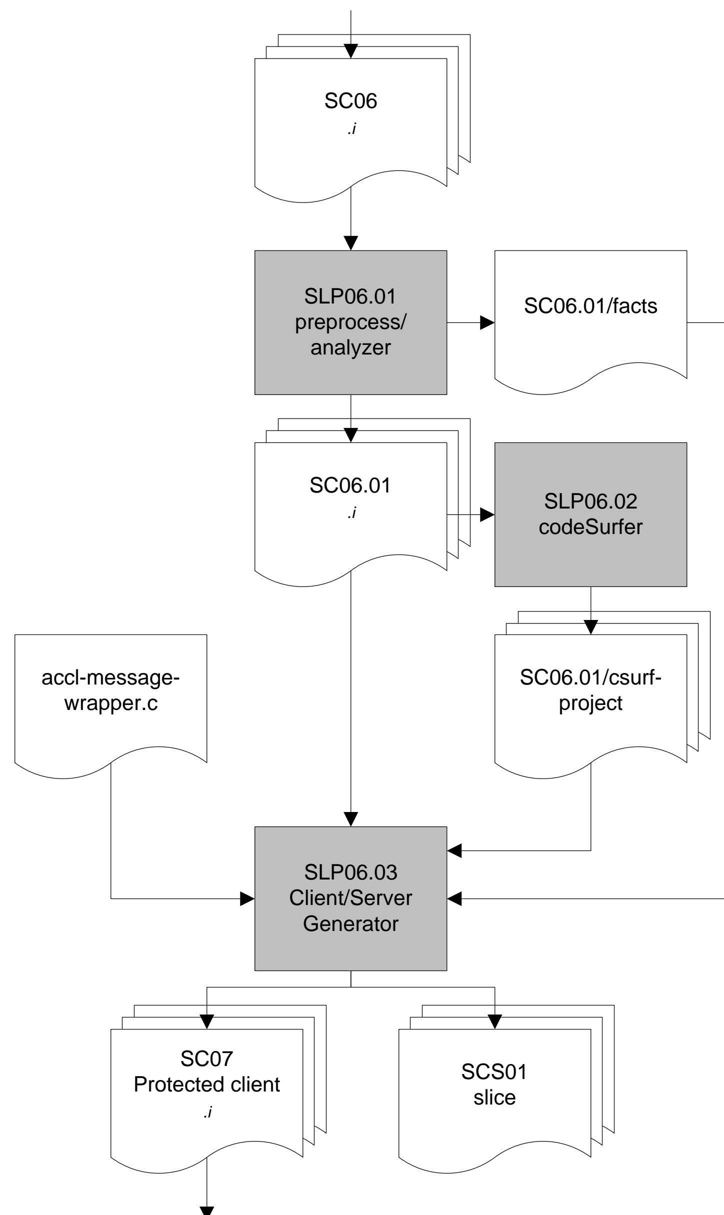

Figure 12: Detailed flow chart of the client-server splitting tool flow in the ACTC

Since M24, the client-server code splitting tool, as described in task T3.1, has been modified to support an intra-procedural slicing algorithm, while previous version was implementing an inter-procedural slicing algorithm. The server-side code that is responsible for running the slice (i.e., the portion of code that is moved from client to server) and the client-side communication library have also been rewritten (for a detailed report on the technique, see deliverable D3.04)

The intra-procedural slicing algorithm has been implemented in CodeSurfer. This algorithm increases the applicability of the tool, and reduces the size of the slice that is generated. This has two main benefits, since the reduced size of the slice minimizes 1) the workload at server-side (the actual code to execute is smaller) and 2) the risk of having incompatibility with the split code and the server-side architecture, in particular for library functions (the client-side code runs generally on arm, while the server-side code runs on x86). The change in the slicing algorithm comes with no negative side effects.

The server-side architecture has been revised in the M30 version of the tool. The client-server code splitting server can handle multiple slices from the same or from different applications. The

tool generates a slice.c file, which has its own slice-handler.c file. The two files can be compiled together to generate a libslice.so component that is loaded by the client-server code splitting server when needed.

Configuration

```
// client server clode splitting
"SLP06": {
  "excluded": false,
  "traverse": false,

  // Process
  "_01": {
    "excluded": false,
    "options" : []
  },

  // CSurf
  "_02": {
    "excluded": false
  },

  // Code transformation
  "_03": {
    "excluded": false
  }
},
```

6.7 SLP08: Offline Code Guards Transformations

Source: Deliverable D5.08 Section 3.2, edited for clarification
Type: Source and binary

The source-level part of this technique consists of two Python tools. The first tool, SLP08_01, rewrites all source code by inserting calls to attestators or verifiers, as instructed by the respective annotations. The second tool, SLP08_02, adds new source code to the application as required, in the form of C files. Those files define the attestators, verifiers, and reaction mechanism. They are then pre-processed (SLP08_03), and added to the application. More details on the code guards protection and the first two tools can be found in deliverable D2.10.

The ACTC has been extended to support this tool by adding an additional source-to-source compilation step called SLP08 as shown in Figure 13. The code guards tool does not require any specific configuration in the ACTC configuration JSON file. From the ACTC's perspective, code guards is a monolithic protection: one shell script invokes the three steps SLP08_01, SLP08_02, and SLP08_03. Only the final files SC08 are actually stored in the build directory of the application being protected.

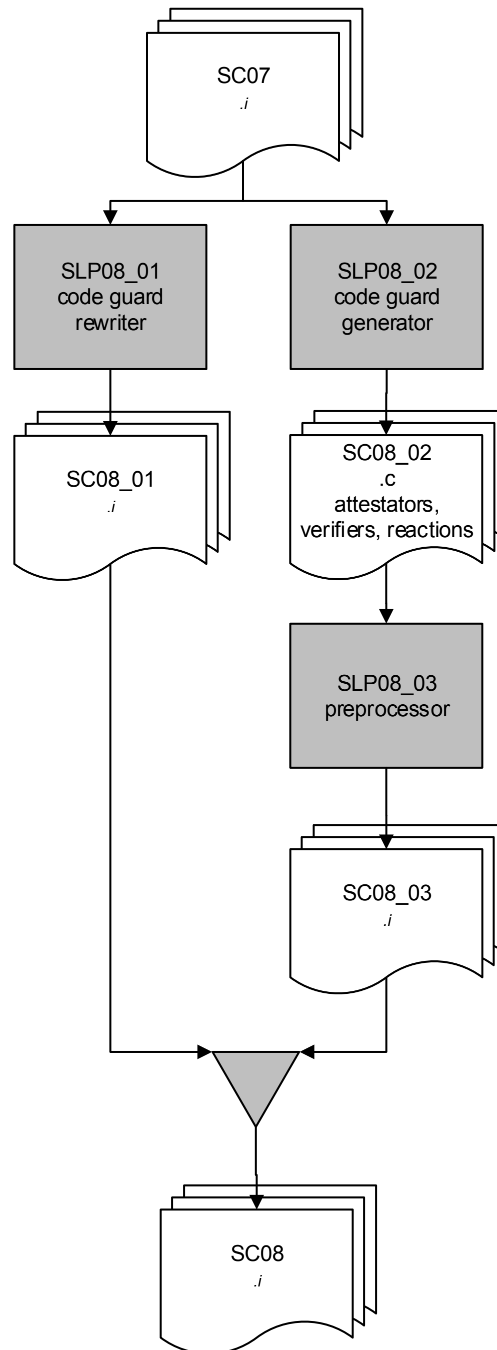

Figure 13: Detailed flow chart of the offline code guards tool flow in the ACTC

Configuration

```
// code guard
"SLP08": {
  "excluded": false,
  "traverse": false,
  "options" : []
},
```

6.8 SLP09: Anti-Cloning Transformations

Source: Deliverable D5.08 Section 3.4, light revision
Type: Source only

To support the anti-cloning mechanism, a dedicated script replaces each anti-cloning annotation by the call to the dedicated anti-cloning function. Moreover, the anti-cloning and the ACCL objects are linked in by the ACTC when anti-cloning is enabled. The complete anti-cloning mechanism has been described in deliverable D3.06.

The ACTC step for invoking the script is the source-to-source compilation step called SLP09 as shown in Figure 14. The anti-cloning tool does not require any specific configuration in the ACTC configuration JSON file.

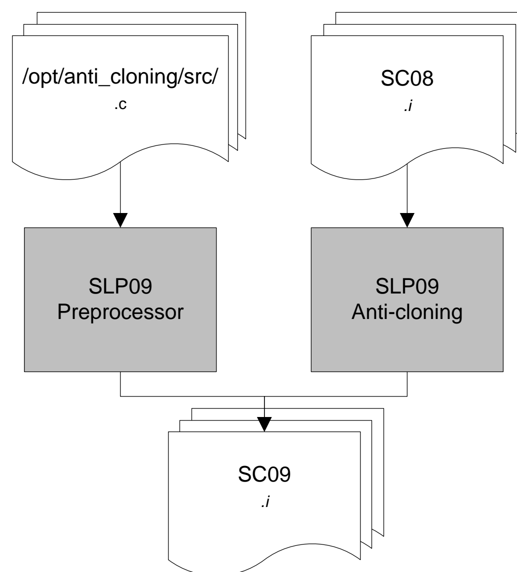

Figure 14: Detailed flow chart of the anti-cloning tool flow in the ACTC

Configuration

```
// anti-cloning
"SLP09": {
  "excluded": false,
  "traverse": false,
  "options" : []
},
```

6.9 SLP10: Reaction Unit Transformations

Source: Deliverable D5.09 Section 3.1, light editing
Type: Source only

The ACTC step SLP10 as shown in Figure 15 supports the reaction mechanism protection tools as a source-to-source compilation step, for which a dedicated shell script is invoked. This shell script processes the reaction-unit annotations. Several supporting files (two python scripts, and a TXL binary) are also added in the ACTC environment to enable the processing of the protection. Several reaction-unit source codes are added, i.e., reaction_waiting.c and reaction_enforcement.c,

with its corresponding header files. These files are where the implementation of reaction-unit is done, including the initialization, the notification, and the triggering of the reaction itself. Details about the reaction-unit mechanism can be found in deliverable D3.06.

When the reaction-unit mechanism is enabled, the ACCL library and the libwebsockets library are linked into the application by the ACTC. There are no specific options in the ACTC configuration JSON file. An overview of the reaction unit tool flow can be seen in Figure 15. To enable processing further down in the ACTC tool flow, the reaction unit source code is pre-processed.

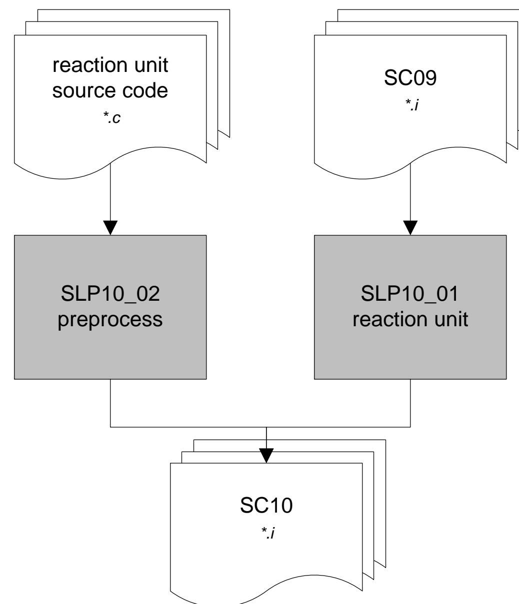

Figure 15: Detailed flow chart of the reaction-unit tool flow in the ACTC

Configuration

```
// reaction unit
"SLP10": {
  "excluded": false,
  "traverse": false,
  "options" : []
},
```

6.10 SLP11: Diversified Crypto Library Transformations

Source: Deliverable D5.09 Section 3.2, unmodified
Type: Source only

The Diversified Crypto tool has been implemented and integrated into the ACTC. A dedicated script has been added to replace each supported annotations. This tool relies heavily on several native libraries for Android to perform the critical cryptographic operations, namely libdcl.so, libisd.so and libmedl.so, shown as the “DCL libraries” in Figure 16. The DCL acronym stands for Diversified Crypto Library. Additionally, several Android assets file are required by the library. All of these files will be copied by the ACTC.

The step SLP11_01 shown in Figure 16 replaces all DCL annotations that are found in the source code by the corresponding calls to the DCL libraries. Input files are taken from the `SLP10` directory and updated files are generated into `SLP11` directory. The purpose of the step SLP11_02 is

to include some wrapper source files into the files to be compiled. In this step the additional files are checked to be sure no error will occur at compilation time. The files are then copied into the `SLP11` directory.

A detailed description of Diversified Crypto protection can be found in deliverable D2.10. As the pre-compiled libraries contain Android specific implementations of required functionality, this protection can only be deployed on Android, not on Linux.

To support the Diversified Crypto protection, a new source-to-source compilation step SLP11 has been added to the ACTC. If this tool is enabled, the curl and openssl libraries are linked into the protected program in support of the protection. There are no specific options in the ACTC configuration JSON file. An overview of the DCL tool flow can be seen in Figure 16.

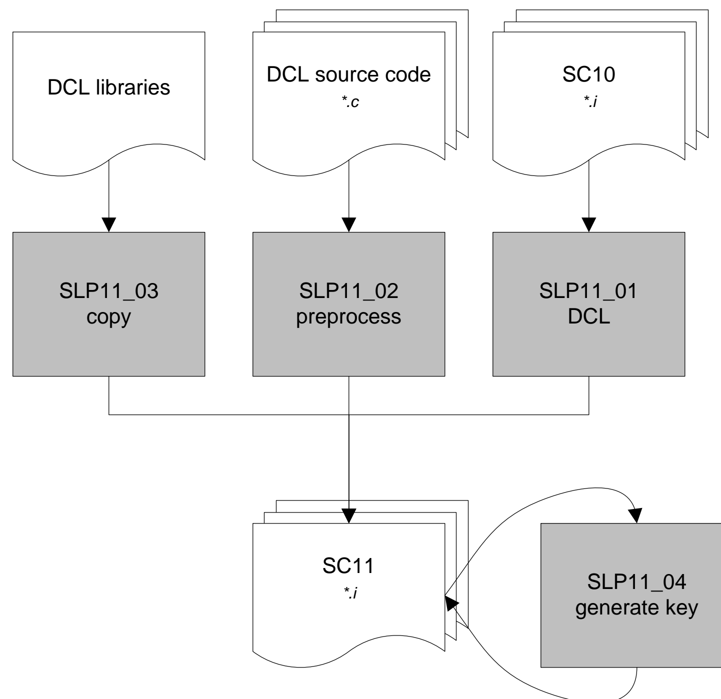

Figure 16: Detailed flow chart of the diversified crypto library tool flow in the ACTC

Configuration

```
// diversified crypto library
// only applicable for ANDROID platform
"SLP11": {
  "excluded": false,
  "traverse": true,
  "options" : []
},
```

6.11 SLP12: Control Flow Tagging Transformations

Type: Source and binary

The CF Tagging (CFT) protection has been implemented and integrated into the ACTC in Year 3 of the project. The CFT protection consists of both a source-level and a binary-level phase. During the source-level phase SLP12, shown in Figure 17, CFT will check the validity of all CFT annotations

in the code, and produce a separate .c file that contains all the verifiers. Two python scripts were developed to do this, along with several C files as a skeleton template. In this phase, however, the annotations are still in the code and not replaced. The generated C files are then pre-processed in phase SLP12_02. Afterwards, during a binary-level phase which is included in BLP04, the actual counter and the verifier will be added at their corresponding places. This is done in a separate module cf_tagging.so that is loaded into the Diablo link-time rewriter and then invoked.

Additionally, CFT supports both offline and online protection variations. They differ with regards to where the verifier is executed. Offline protection means it is executed locally in the application, because the verifier code is embedded within the application code. On the other hand, online protection executes the verifier remotely in the server of the ASPIRE portal. The remote verifier is formed as an .so file that is produced by step SLP12_03, which is only made available if at least one remote verifier annotation exist.

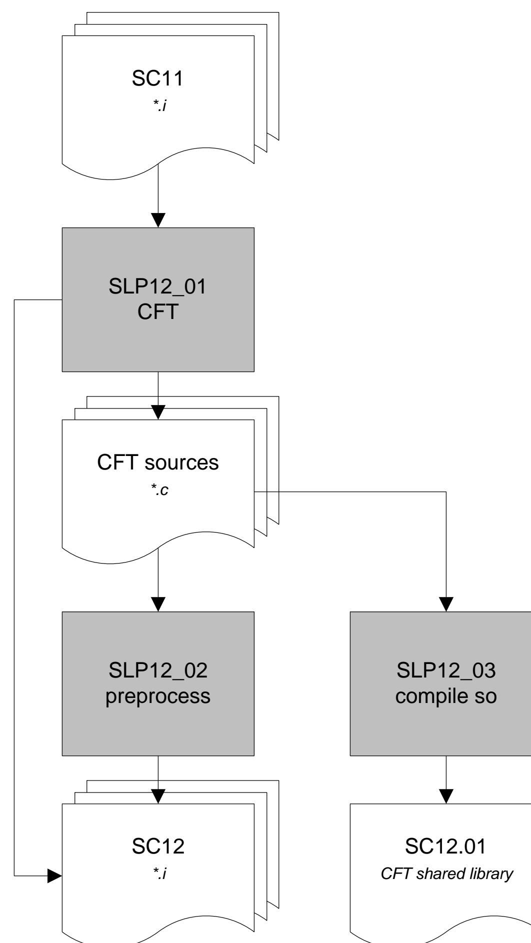

Figure 17: Detailed flow chart of the control flow tagging (CFT) tool flow in the ACTC

Configuration

```
// control flow tagging
"SLP12": {
  "excluded": false,
  "traverse": false,
  "options" : []
}
```

6.12 SLP04: Annotation Extraction

Source: Deliverable D5.01b Section 7.5, updated to reflect latest release
Type: Source only

Once all the source code transformations (apart from remote attestation) are applied, component SLP04 can extract all the annotations from the pre-processed source code. The extracted annotations will then drive the binary-level part of the ACTC.

All information encoded in the annotations is extracted and stored in an annotation fact file D01. For each annotation, the JSON-formatted file D1 contains its line number information (i.e., the line number of the BEGIN annotation and the END annotation, as well as some other auxiliary information that helps in identifying the annotated code. This information includes the file name, the procedure name, and the line number range.

In the binary-level processing tools, the source code line number information will be mapped onto the assembly line number information (i.e., instruction addresses) by means of the DWARF debugging information that the standard compiler inserts into the object files, and that Diablo extracts again.

Besides the annotations extracted from the source code of the application to be protected, users of the ACTC can also directly provide protection descriptions to the binary-level processing phase. They can do so by extending the already mentioned annotation fact file D01 with an external file. That file uses exactly the same JSON format as D01. In the external annotation files, function names and object file names can include wildcards in the form of an asterisk ($*$). Moreover, line numbers can be omitted for each annotation. The latter feature, together with the wildcards allow a developer to specify a binary-level protection for a range of functions or object files at once. This is very handy, for example, to specify that anti-callback checks or control flow obfuscation should be deployed on all linked-in libraries, or that all the code in those libraries should be guarded by code guards.

The annotation extraction work-flow is depicted in Figure 18.

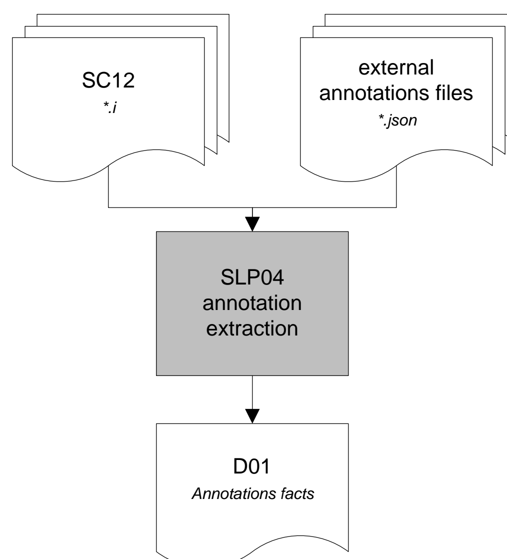

Figure 18: Detailed flow chart of the annotation extraction tool flow in the ACTC

Configuration

The extraction section in the ACTC configuration file allows the user to specify multiple external annotation files:

```
// annotation extraction + external annotation file(s)
"SLP04": {
  "excluded": false,
  "options" : [],
  //External annotation JSON files
  "external": ["../external_annotations1.json",
               "/path/to/external_annotations2.json"]
},
```

6.13 SLP07: Remote Attestation Transformations

Source: Deliverable D5.06 Section 3.5, D5.08 Section 3.5, merged
Type: Source and binary

Remote attestation techniques have been developed and the tool support to insert the necessary remote attestation code functionality into the application to be protected has been added to the ACTC. Remote attestation does not change any of the original and previously inserted source code. Moreover, it can be used to monitor the integrity of code inserted by other protections, thus it is the last step of the source code transformations. Only static remote attestation is supported which has already been presented in Section 5 of deliverable D3.02, and Section 5 of deliverable D3.04. Dynamic remote attestation has been tested but it is still not integrated in the ACTC because of the limitations of the trace extractors that do not allow Daikon to infer invariants on ARM platforms.

The annotation parser is able to properly extract the static remote attestation annotations (and dynamic remote attestation as well).

The ACTC integrates a first remote attestation tool in the source code processing step SLP07, another step is performed in the binary-level processing, which is discussed in Section 8.6. SLP07 is composed of multiple sub-steps, as depicted in Figure 19.

SLP07 is invoked after all other source-level protections have been applied. SLP07 is executed as a Bash script. SLP07 selects the attestators to add into the application as part of the protection. Multiple attestators can be added into the application to protect. The Attestators to insert are specified in the source code annotations. If an annotation does not specify the Attestator to use, two case are considered. If no annotations specify an Attestator, a default Attestator is inserted. If only one Attestator is required by annotations that explicitly specify an Attestator, this Attestator is also used for other annotated code areas that do not specify Attestators. However, if two or more different Attestators are specified by annotations, the RA tool stops and reports an error, as it is not clear to which Attestator to assign the areas to monitor.

In order to support multiple attestators and avoid symbol names conflicts during linking phases, remote attestation source code contains placeholders in globally accessible symbols, which can be replaced before compilation to avoid naming conflicts. Each global symbol name is made unique by appending to its name a unique random alphanumeric string. An example of parametric global symbol is the following one:

```
RA_RESULT ra_prepare_data_NAYjDD3l2s(RA_table table)
```

In this way, all the source code is usable also without characterizing the parametric names, that is useful for server side component compilation. In fact, each extractor and the verifier are directly

associated to only one attestator, thus they do not suffer any naming conflict because they include one and only one implementation of every remote attestation fundamental block. The remote attestation modular structure, based on fundamental building blocks, is described in deliverables D3.02 and D3.04. The latest report on remote attestation can be found in deliverable D3.06. Server side components do not suffer the names conflicts but need to include same source code files of the client side components.

The tool is called as follows:

```
attestator_selector.sh  -o output_dir
-a annotations_json_path
-t target_arch
```

where:

- `output_dir` is the path of the folder where the RA tool produces its output;
- `annotations_json_path` is the path to the JSON file that contains remote attestation annotations to be processed;
- `target_arch` is the name of the target environment; it can be either `linux` or `android`.

The input the JSON file contains the extracted annotations (the D01 annotation facts). The RA tool outputs two object files that will be linked in the protected by the ACTC.

The RA tool work-flow is the following one.

1. In the first phase, the tool invokes the annotation interpreter that extracts the information needed to include the proper implementation files according to what is specified by the annotations. The interpreter produces two files per attestator.
   (a) The attestator description, this file is named as the attestator label (defined in the annotations) and contains the reference to implementation file of the remote attestation modules for the attestator (as requested by the annotation). This file contains a set of variable definitions following the syntax used by makefiles. Each variable definition specifies the name of the real implementation file (.c) of the associated remote attestation fundamental block. The content of this file is, for instance:

   ```
   RA_DATA_PREPARATION_BLOCK_NAME := ra_data_preparation
   RA_DO_HASH_BLOCK_NAME := ra_do_hash_sha256
   RA_NONCE_INTERPRETATION_BLOCK_NAME := ra_nonce_interpretation_3
   RA_NONCE_GENERATION_BLOCK_NAME := ra_nonce_generation
   RA_DATA_TABLE_BLOCK_NAME := ra_data_table
   RA_MEMORY_BLOCK_NAME := ra_memory
   ```

   (b) The second file, the attestator frequency file, contains the required attestation frequency, as it is interesting only for the server-side logic. It is named `attestator_label.freq`.

2. Starting from the files produced in the first step, the RA tool begins the characterization phase. For each attestator descriptor, the tool selects the specified remote attestation fundamental blocks files and characterize the string placeholders with the attestator label. The output of this step consists in a folder per required attestator. Each of those folders contains all the characterized source code needed to produce the relative attestator binary. In addition to the fundamental blocks source files, the output folder will contain also all the source files that do not need any customization and that must be included in the attestator binary compilation. At the end of this phase, the output folder contains as many new folders as the number of the defined attestators. Each attestator folder will look like:

   ```
   output_folder/tmp_first_attestator
   |-- attestator.c
   |-- attestator.h
   |-- ra_data_preparation.c
   |-- ra_data_preparation.h
   |-- ra_data_table.c
   |-- ra_data_table.h
   |-- ra_do_hash.h
   |-- ra_do_hash_sha256.c
   |-- ra_memory.c
   |-- ra_memory.h
   |-- ra_nonce_generation.c
   |-- ra_nonce_generation.h
   |-- ra_nonce_interpretation_3.c
   |-- ra_nonce_interpretation.h
   ```

   Each global parameterized symbol is transformed, for example, from

   `RA_RESULT ra_prepare_data_NAYjDD3l2s(RA_table table)`

   is transformed into

   `RA_RESULT ra_prepare_data_frist_attestator(RA_table table)`

3. Then, the RA tool runs the proper compiler (it depends on the specified target architecture) in each folder produced by the previous step to generate an object file per attestator. All the Attestators object files are then linked together to produce a unique attestators.o object file that is delivered to the ACTC to be linked as part of the protected application. The `attestators.o` is delivered along with a pre-compiled object file, named `racommon.o`, in the output directory specified as input option for the tool. The `racommon.o` file contains a collection of functions which are used by all the attestator.

The overall remote attestation tool work-flow is depicted in Figure 19.

The RA tool is portable and can be executed on any platform where a Bash shell and Java (at least version 1.5) are installed. At the current stage of development, the remote attestation tool can be used with the examples provided by the ACTC maintainers, the NAGRA and SFNT use cases, and other open source applications used to test this technique and to other techniques that build on the static remote attestation (e.g., reactive attestation).

As explained in Section 8.6, a second remote attestation processing step is implemented in the binary-level part of the ACTC, in BLP04. In that step, the Diablo-based rewriter determines the exact location of the code areas to attest in the memory space of the protected binary or library.

The tool supports annotations logging. Further details on the static remote attestation tool are available in deliverable D3.04, Section 5.3.3.

Moreover, the RA tool supports the annotations that contain the `attest_at_startup` feature (as described in D3.09). By flagging a code area with the `attest_at_startup` feature, the Attestators

that have to protect that code regions verify its integrity as soon as the application is launched. The Remote attestation server side components implement a stateful session mechanism (that declares an application safe if and only if all the `attest_at_startup` code areas of have been verified) and the ASPIRE database stores data to keep track of the currently active sessions.

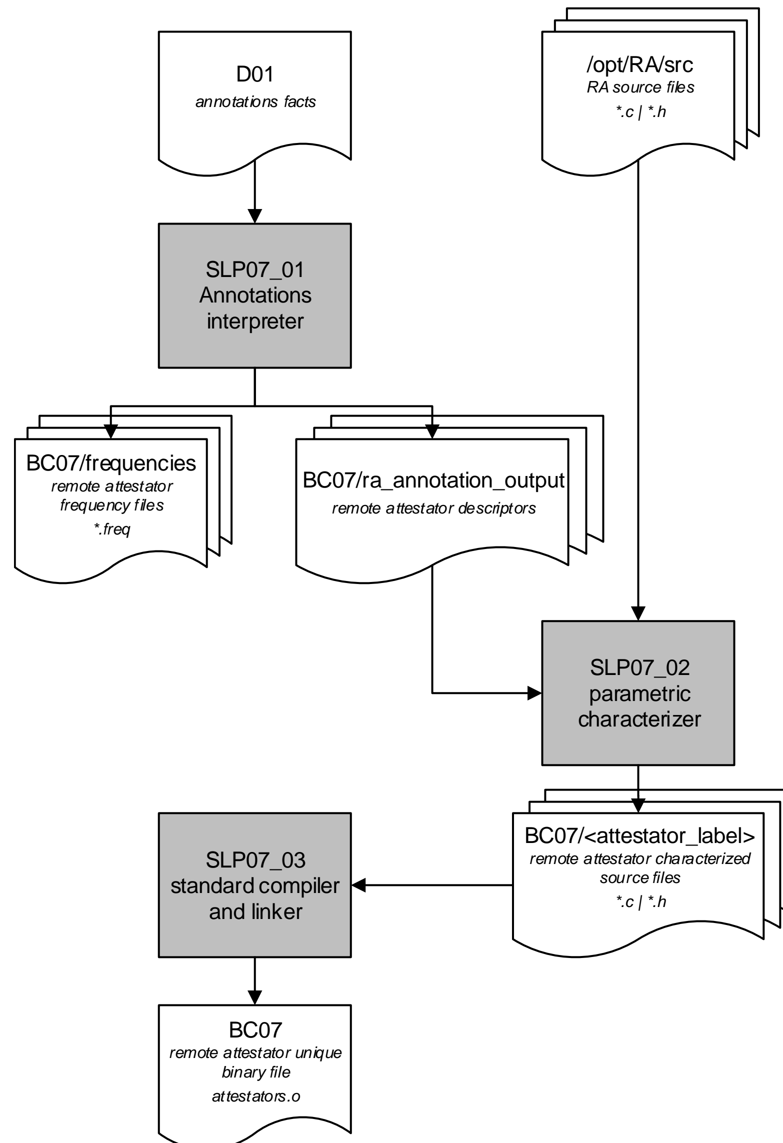

Figure 19: Detailed flow chart of the remote attestation tool flow in the ACTC

Configuration

```
// remote attestation
"SLP07": {
"excluded": false,
"options" : []
},
```

7 Compiler, Assembler and Linker

Source: Deliverable D5.01b Section 8, updated to reflect latest release
Section authors:
Bjorn De Sutter, Jeroen Van Cleemput (UGent)

7.1 Compiler Requirements

The source code files SC12 of the application protected at the source level are compiled by a "regular" compiler. The generated assembler files are then be assembled by a "regular" assembler, and the generated object files are linked into binary executables or dynamically linked libraries by a "regular" linker.

The term "regular" above denotes that with the ACTC, we in theory aim to support any compiler, assembler, and linker that behaves in such a way that the generated libraries or binaries can be rewritten conservatively at link time. In the past this capability has been demonstrated with the Diablo link-time rewriting framework from UGent, for code generated with several generations and brands of existing compilers, both proprietary (e.g., with ARM ADS, ARM RCVT, ARM RVDS, Microsoft Visual Studio) and open source (e.g., with GCC, LLVM, and binutils).

More concretely, the support for conservative link-time rewriting depends on the availability in the object files of enough symbol information and relocation information. Sufficient such information needs to be present to disassemble the binary code correctly, and to overestimate the potential indirect control flow transfers in the binary code in such a way that the overestimation is sound and precise enough not to prohibit useful transformations on the code.

Some compilers, assemblers and linkers provide sufficient information by themselves, such as those in the proprietary ARM RVDS tool chains. Others do not provide sufficient information, typically because they are designed to support regular linking only — not link-time rewriting, and because shortcuts can then be taken that reduce, e.g., the file size of the object files or libraries.

For recent versions of the open-source compilers GCC (v4.8.1) and LLVM (v3.4) and for recent versions of the assembler and linker of binutils (v2.23.2), a series of (small) patches was developed at UGent to make them provide enough information to the link-time rewriter Diablo, which is used in the binary-level protection tools in ASPIRE.[2] In addition, some minor patches were made to integrate LLVM in the crosstools tool (http://crosstool-ng.org) that we use to configure and build the compiler, assembler and linker in the early version of the ACTC. All of these patches are listed in tables 1, 2, and 3.

Diablo currently does not handle exception handling code and exception handling data correctly (when non-trivial transformations are applied). Complete support for exception handling is foreseen for after the ASPIRE project. In the mean time, Clang-LLVM and GCC need to be invoked with the flags `-Wl,--no-merge-exidx-entries`, and only applications or libraries that do not depend on exception handling can be processed.

clang.patch	generate $handwritten mapping symbols around inline assembler
crosstool-fix.patch	patch reused from the clang-crosstools fork to integrate clang with crosstools binutils (https://github.com/diorcety/crosstool-ng)

Table 1: Patches to Clang - LLVM

[2]The mentioned versions of the compilers and binutils were "recent" at the start of the ASPIRE project. During the project, the partners focused on developing and integrating new protections, not on investing non-scientific engineering effort in updating their tool flow to support more recent versions.

annotate_handwritten_asm.patch	patch to generate $handwritten mapping symbols around inline assembler
disable_tm_clone.patch	patch to disable the generation of sections related to transactional memory
fix_parallel_build.patch	patch to allow parallel build of the compiler, obtained from crosstools mailing list)
enable_dwarf_crtbeginend.patch	patch to omit debug information in crt object files (such that binutils' link-once support works properly on code generated with all of the above patches)

Table 2: Patches to GCC

remove_eh_frames.patch	omit exception handling frames (which are not supported yet in Diablo, support is foreseen in the future)
disable-more-merge-eidx.patch	disable exidx section merging
mark_code_data_sections.patch	generate mapping symbols that mark data in code sections
disable_section_merge.patch	disable section merging during linking
disable_relaxation.patch	disable symbol relaxation
add_relative_symbols.patch	generate additional mapping symbols that allow Diablo to relocate code more aggressively
fix-neon-vshll-qd.patch	back-ported patch from later binutils to fix objdump NEON instruction disassembler

Table 3: Patches to binutils

7.2 Compilation and Linking Tool Chain

Figure 20 depicts the use of a regular compiler, assembler and linker in the ACTC. All source code files are compiled with Clang - LLVM 3.4 and are assembled into object files (BC08) using the GNU assembler (as) found in binutils, and linked using the GNU linker (ld), also part of binutils. The result (BC02) is either a binary executable a.out or a dynamically linked library liba.so. The ACCL, if it is required to be linked into the protected application for the online protection techniques, is also compiled from source and linked into the binary or dynamically linked library. This compilation is done by the ACTC, i.e., together with the compilation of the software to be protected, because through the ACCL, features such as the ASPIRE server IP address, the unique ASPIRE application ID, etc. are embedded into the protected application. Whether or not the ACCL is linked in is determined on the basis of the annotations that were extracted from the source code and that were stored in D01.

Similarly, based on the annotations, the ACTC decides to link-in some pre-compiled objects and libraries, if necessary, to implement code mobility (a binder and a downloader), renewability (which builds on code mobility), and supporting libraries used by the ACCL, such as `libcurl.a`, `libssl.a`, `libwebsockets.s`, and `libcrypto.a`.

The linker is invoked with the -Map flag to produce a so-called map file (BC02) that logs the operation of the linker.

Configuration

The `src2bin` section in the ACTC configuration files allows the developer to specify preprocessing options (as already discussed in Section 6.4), compilation flags, linker flags, and necessary information to be embedded in the ACCL in case it is linked into to the software in support of online protections:

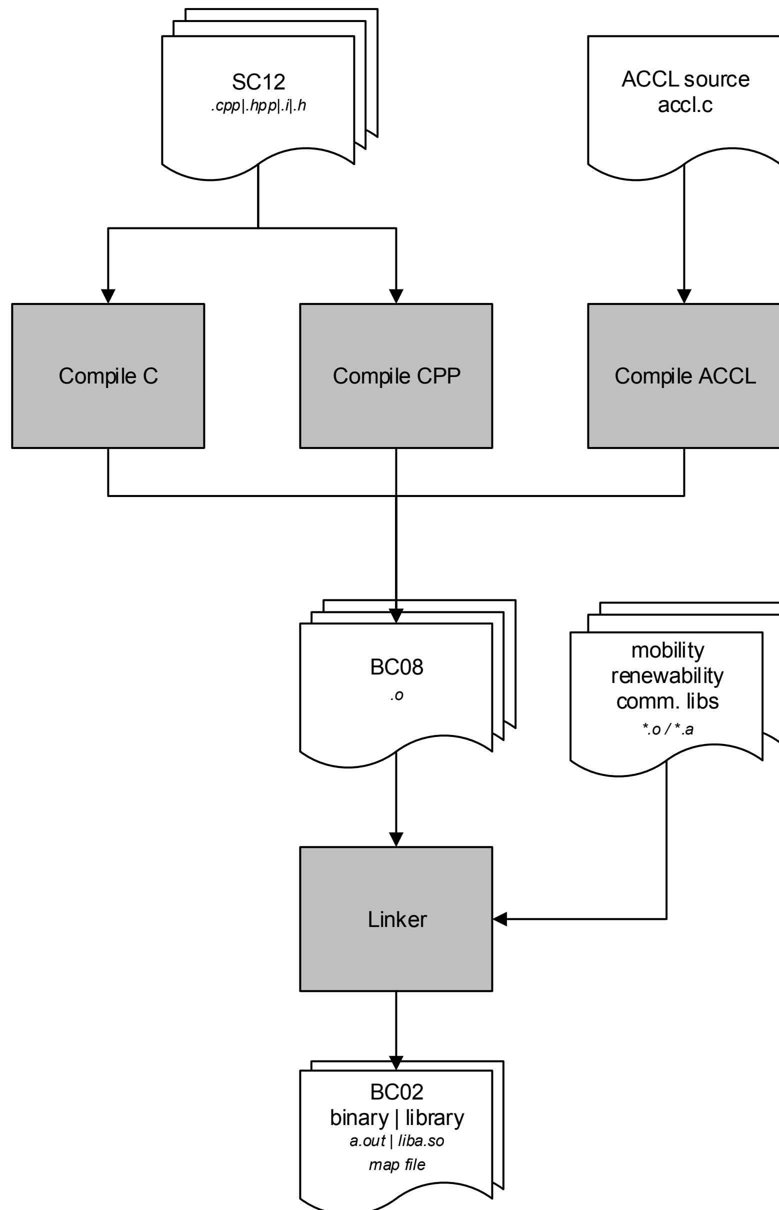

Figure 20: Compiler and linker part of the ACTC.

```
// Assembler, Compiler, Linker
"src2bin": {
"excluded": false,
  // Common options for all tools
  "options"       : [],

  //Preprocessor options for SLP02 tool
  "PREPROCESS": {
    // -I <dir>
    // -isystem <dir>
    // -include <file>
    // -D<macro[=defn]>
    "options"     : []
  },

  // .c, .cpp
  "COMPILE": {
    //Common compilation options
    "options"     : [],
    //C compilation options
    "options_c"   : [],
    //CPP compilation options
    "options_cpp": []
  }
```

8 Binary Rewriting Tool Chain

Section authors:
Bjorn De Sutter, Jeroen Van Cleemput, Bert Abrath (UGent)

8.1 Overall Binary Rewriting Approach

The overall binary code protection approach in ASPIRE consists of four major steps, as depicted in Figure 21.

In the first step, BLP01, the binary code is analyzed to decide where and how to apply the binary-level protections that require the generation and integration of additional custom software components.

In the first step, BLP01, native code is identified in the binary that needs to be replaced by bytecode for the client-side code splitting (SoftVM) approach (see D1.04 Section 3.1).

The generation of the customized bytecode and of the customized SoftVM that can interpret the bytecode (in the form of object files BC03) constitutes the second step BLP02 of the binary-level part of the ACTC.

In the third step, BLP03, the custom components of BC03 are integrated: they are linked into the application, together with some fixed and pre-compiled software components if those are needed for the other protections (as indicated by the annotation facts) and with remote attestators if those have been generated in support of remote attestation. The result are a binary/library BC04 with a map corresponding linker map file.

Then, as the first processing step in BLP04 the original code of the linked application BC04 is rewritten to actually invoke the linked-in SoftVM on the embedded bytecode fragments: for client-side code splitting, the previously selected native code fragments are replaced by stubs that invoke the linked-in SoftVM to interpret the corresponding, linked-in bytecode fragments.

Later in the fourth step BLP04, all rewritten code and all integrated components are further protected by applying all the other binary-level protections, incl. obfuscation, anti-tampering, and anti-debugging protections. Finally, the final code layout is determined, and the code is assembled and relocated (when necessary). At that point, place-holders that might have been inserted during the rewriting in steps two, three, and four can be filled in. All binary-level transformations, as they are deployed, are extensively logged in log files, incl. files that visualize the control flow fragments being transformed, before and after each transformation step.

8.2 Diablo

The tool that will be used for the binary-level code analyses, transformations, and protections, is Diablo. This link-time rewriting framework has been under development at Ghent University for about 15 years. For different purposes (such as speed optimization, code compaction, code protection), different front-end tools have been built on top of this framework.

8.2.1 Basic Diablo Operation

Figure 22 depicts the most important inputs and outputs of a typical Diablo front-end. The inputs consist first of all of the native code file of the application to be rewritten. This can be a (statically or dynamically linked) binary (like a.out), as well as a dynamic library (like liba.so). To correctly rewrite this native code, symbol information and relocation information are obtained from the original object files that were linked into the application by the original linker. In the case of statically linked binaries, also the object files coming from the statically linked-in libraries (like libs.a) need to be retrieved to collect their symbol information and relocation information. Furthermore,

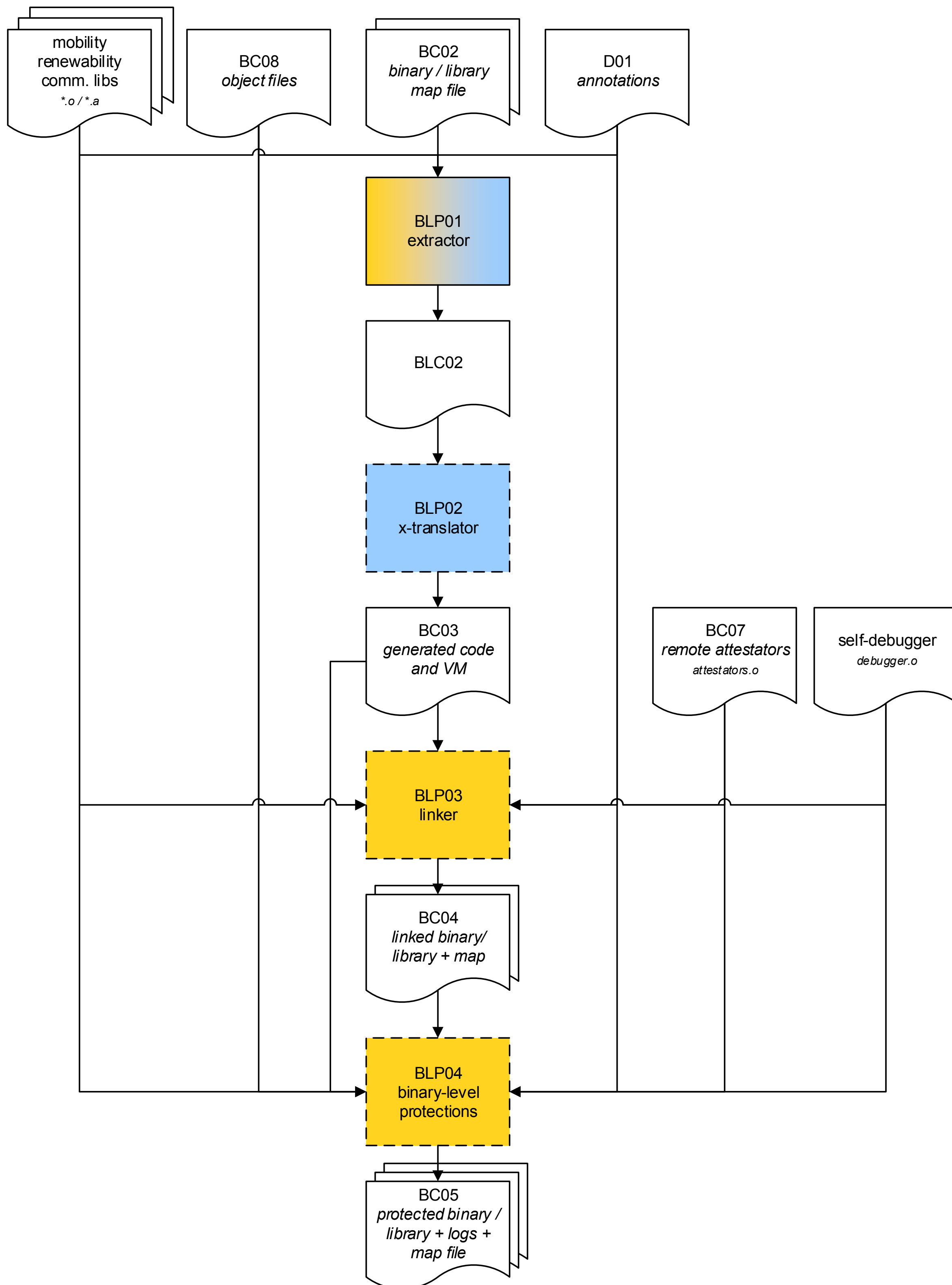

Figure 21: Four steps of the binary-level part of the ACTC

Diablo needs to know how the original linker linked all the object files into the application. All relevant information thereto can be found in the linker script, which describes, the operation of the original linker in general terms and which comes with Diablo, and in the so-called map file that the original linker produced when it generated the original application. Optionally, Diablo can also obtain profile information consisting of basic block execution counts.

Based on this input, a Diablo front-end tool goes through a number of phases:

1. **Linker Emulation**: First, Diablo links the original object files again, emulating the behavior of the original linker as indicated by the linker script and the map file. Following this linking, Diablo compares the linked code to the code in the input application. When there are mismatches, Diablo halts, informing the user that it apparently was not able to interpret the provided relocation information or symbol information correctly. So besides the actual collection of this information, this step also serves as a validation of the information.
2. **Disassembly**: Diablo disassembles all the code in the application.
3. **Control Flow Graph Reconstruction**: Diablo partitions the code and data of the application in chunks: basic blocks for the code, which are further partitioned into procedures, and blocks for the data. All of them are incorporated into a big graph representing the program call graph, the procedures' control flow graphs, and all pointer-references between code and data blocks.
4. **Analysis and Transformations**: The code analyses and transformations are performed on the graph as specified by the front-end tool. These transformations can take into account the profile information when it is available.
5. **Code & Data Layout**: The blocks in the graph are linearized, i.e., put in a specific order. This can also take into account the profile information to minimize code size and to optimize the instruction cache behavior.
6. **Assembly**: The code is assembled again.
7. **Code Generation**: The rewritten, final binary (here called b.out to denote it is the rewritten version of a.out) or library (here similarly called libb.so) are produced.

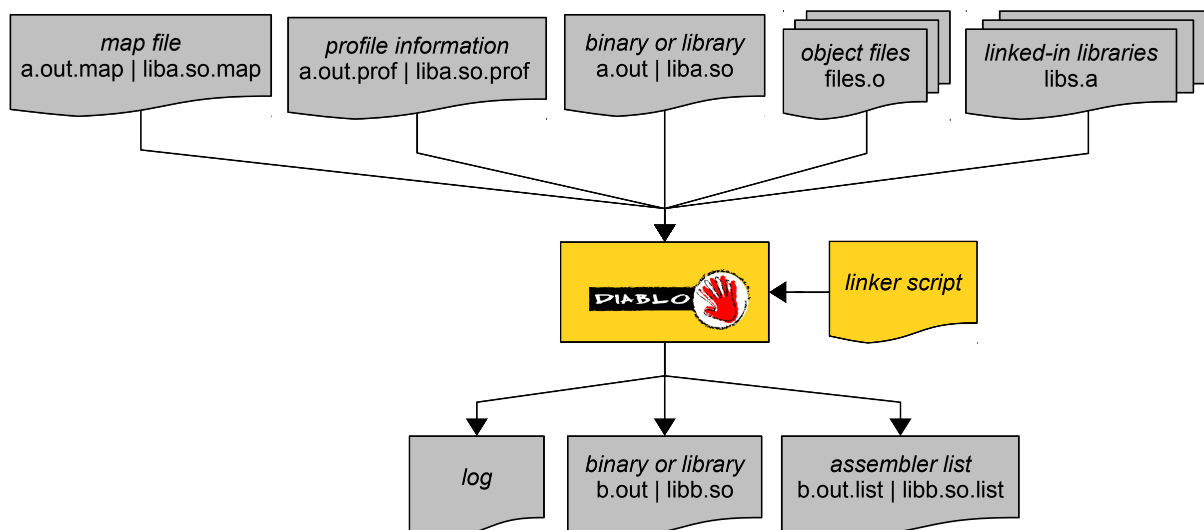

Figure 22: Inputs and outputs of Diablo.

Together with the produced application, Diablo also produces a corresponding list file. This is a disassembled version of the generated program with some additional information, such as the address of each instruction in the original application when that instruction originated from that application, i.e., when it was not injected by Diablo as part of some transformation. Finally, Diablo produces a log. Depending of the level of verbosity chosen upon invocation of Diablo, different levels of logging can be produced.

8.2.2 ASPIRE-specific Diablo Development

Since the start of the project, Diablo has undergone a major development effort to prepare it for the use cases and target demonstration platform of ASPIRE. The major developments regarding support for the supported compiler versions are the following:

- flow graph support for jump table instruction sequences as generated by recent versions of gcc and LLVM;
- support for handling more aggressively scheduled code (in which, e.g., instructions involved in the computation of relocatable addresses are scheduled in between other instructions and possible even span procedure calls);
- improved support for more modern glibc features such as thread-local storage;
- support for relocations as generated by the more recent binutils;
- support for additional mapping symbols generated by binutils;
- numerous fixes for bugs that got triggered by deploying Diablo on code generated by the new compilers, i.e., both on the application code of the SPEC2006 benchmarks, as well as on the code of the more recent eglibc.
- support for ARM Erratum #657417 on early revisions of Cortex-A8 processors;

With respect to the ARM architecture, a considerable extension has been implemented. Before this project, only the ARMv4 + some ARMv6 instructions were fully supported in Diablo, i.e., Diablo could disassemble and assemble them, construct and layout control flow graphs, and apply all its optimizations on the code. Today, the ARM support in Diablo has been extended as follows:

- disassembler and assembler support has been developed for all ARM NEON, VFP, Advanced SIMD and user-space ARMv7 instructions;
- control flow graph construction and code lay-out support has been developed for all ARM NEON, VFP, Advanced SIMD and user-space ARMv7 instructions;
- we have extended the existing analyses and optimizations to handle the ARM NEON, VFP, Advanced SIMD and user-space ARMv7 instruction set correctly, incl. but not limited to, support for
  - single, double, and quad registers in VFPv3;
  - support for the ARMv7 instructions MOVW and MOVT that replace address-pool based generation of absolute addresses;
  - numerous instructions that only overwrite part of their destination registers (needed for liveness analysis);
  - instructions that operate on different data widths and vector types (needed for factoring and peephole optimizations);
  - floating-point and vector memory operations that write back the stack pointer (needed for load-store forwarding as well as stack frame optimization);
  - code layout and address producer optimization has been updated to take into account alignment requirements of 64-bit, 128-bit and 256-bit accesses to data pools in the code section.

Before this project started, Diablo only supported the rewriting of statically linked binaries. Today, Diablo also supports dynamically linked ELF binaries and dynamically linked libraries: for the supported compiler version (LLVM 3.2 - 3.3 - 3.4, GCC 4.6.4 - 4.8.1, binutils 2.23.2) and standard library versions (eglibc 2.17) all necessary relocations, symbols, section types, etc. are now supported. The existing analyses, optimizations and obfuscations in Diablo all work on the statically linked as well as on the dynamically linked binaries and libraries. The developed functionality includes

- support for modeling and maintaining multiple entry points;
- extended and cleaned up support for dynamic relocations and dynamic symbols;
- support for position-independent code analysis, optimization, and generation;
- support for more complex scenarios involving GOT and PLT entries;
- support for correctly resolving and generating versioned symbols.

8.3 Client-Side Code Splitting (SoftVM)

Client-side code splitting (as designed in D1.04 Section 3.1) is implemented in the three first steps BLP01–BLP02–BLP03 discussed in Section 8.1 and in the first step of BLP04.

8.3.1 BLP01: Native Code Extraction

As indicated in Figure 23, in BLP01.01 a Diablo rewriter collects the code fragments that need to be translated from native code to bytecode. It does so on the basis of the annotation facts D01 assembled by the source-level component SLP04, and based on its usual inputs, which in this case correspond to the application BC02 to be rewritten, the corresponding map file (D02), the object code (BC08) and (optionally) the pre-compiled code that was linked into the original application by the standard linker (as described in Section 7.2).

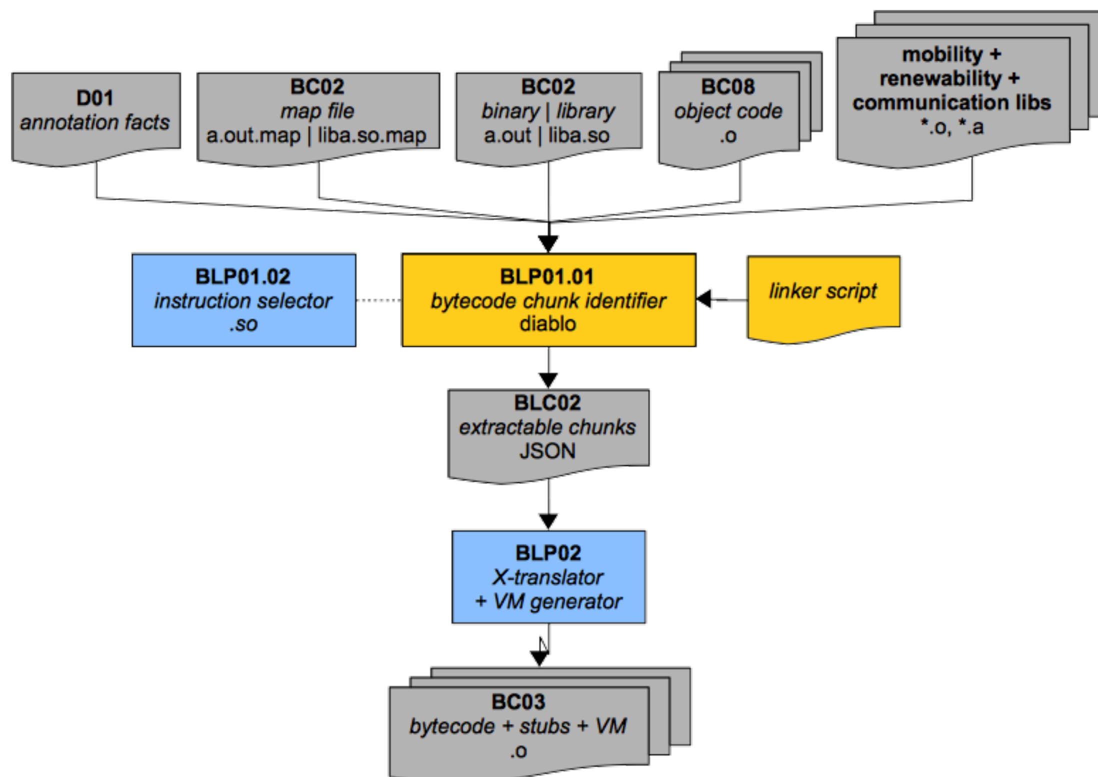

Figure 23: Tool flow components for chunk extraction and bytecode generation

Diablo produces a description of the native code chunks in the form of a JSON file (BLC02). The specification for this interface is presented in Appendix C.

To select the native code fragments to be translated to bytecode, the Diablo tool considers code regions marked as such in the annotation facts D01. Within these regions, all possible fragments are be selected, i.e., all fragments of which the instruction selector indicates that the instructions in them are supported by the X-translator and the SoftVM.

Configuration

```
// Native Code Extraction
"BLP01": {
  "excluded": false,
  "traverse": false,
  "options" : []
},
```

8.3.2 BLP02: Bytecode Generation

The second tool BLP02 in support of client-side code splitting is the X-translator. Based on the JSON file of BLC02 it generates bytecode, as well as stubs that will replace the selected native code fragments. The responsibility of the stub is to invoke the SoftVM that will be embedded in the application in BLP03, to let it interpret the generated bytecode that replaces the original native code, as well as to pass the program state to the SoftVM before its invocation.

The stubs and the bytecode will be generated as code and data sections in ELF object files, that can simply be linked into the application to protect.

UGent was responsible for the code extraction in the Diablo rewriter, while SFNT was responsible for the X-translator (as well as the SoftVM). This separation of concerns ensures a clear separation of Foreground IP, and a tool flow design in which components can easily be replaced by alternative ones after the project to facilitate exploitation of the project results.

Renewability

Support for diversified bytecode, as described in deliverable D3.08 has been added to the the ACTC in version 2.6. To this end, the X-translator tool BLP02 operates as shown in Figure 24. Source code for the VM is generated by the X-translator tool together with the generated (and customized) bytecode and stubs. The generated C code is stored in BC03/out_gen_vm. This source code is then compiled by the standard ACTC compiler and archived in the vm.a file.

An additional bytecode_diversity seed configuration option has been added to the ACTC configuration JSON to seed the randomness of the generated bytecode. The option can be set to any (32bit) integer value or 'RANDOM'. During compilation, the seed is passed to both the X-translator (BLP02) and the Diablo tool that performs the bytecode & VM integration and all other binary-level obfuscations (BLP04).

Configuration

```
// Bytecode Generation
"BLP02": {
  "excluded": false,
  "options" : []
},
```

The random seed for renewable bytecode can be configured using the *bytecode_diversity_seed* option in the *bin2bin* section of the ACTC configuration JSON:

```
// bytecode diversity seed, integer or RANDOM
"bytecode_diversity_seed"     : "0",
```

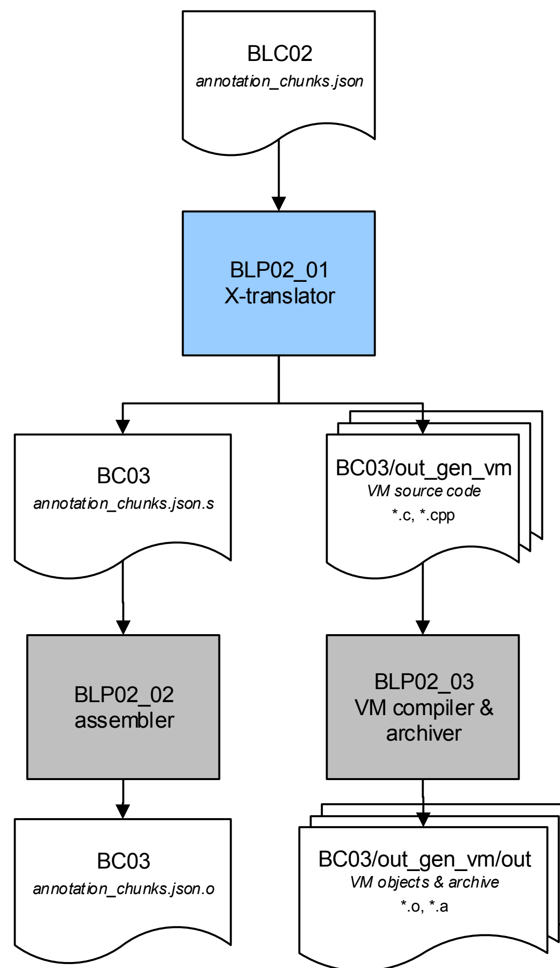

Figure 24: Tool flow components for bytecode generation

8.4 BLP03: Code Integration

As can be seen in Figure 21, BLP03 is in essence an extra linking step in which all existing code, both of the original application as it has been transformed at source level and then compiled, and of the components needed to implement various protections, is linked together. This produces a new binary or library in directory BC04, together with the corresponding map file.

Configuration

```
// Code Integration
"BLP03": {
  "excluded": false,
  "options" : []
},
```

8.5 BLP04 - Part 1: VM Invocation & Relocation Fix-ups

Finally, Figure 25 shows the last compilation/rewriting step, in which a second tool BLP04 based on Diablo rewrites that linked library/binary of BC04 to finalize the client-side code splitting protection. To that extent, this tool first replaces the native code fragments that have been translated

by the X-translator in step 2 by control flow transfers to their corresponding stubs. UGent was responsible for implementing this rewriting step in this integration in the Diablo tool.

At the very end, the Diablo rewriting tool interfaces again with the instruction selector and with the X-translator. This is done to regenerate the bytecode such that any code addresses hard-coded and encoded in that bytecode match the intended addresses in the final, fully protected binary/library that is produced in BC05. By invoking the instruction selector and the X-translator again, as shown in Figure 25 rather than letting Diablo perform the relocations of those addresses in bytecode, the separation of concerns is maintained: one partner, SFNT, is the sole responsible for deciding on how to encode those addresses in the binary, and for on how to decode and use those addresses when the VM interprets the bytecode.

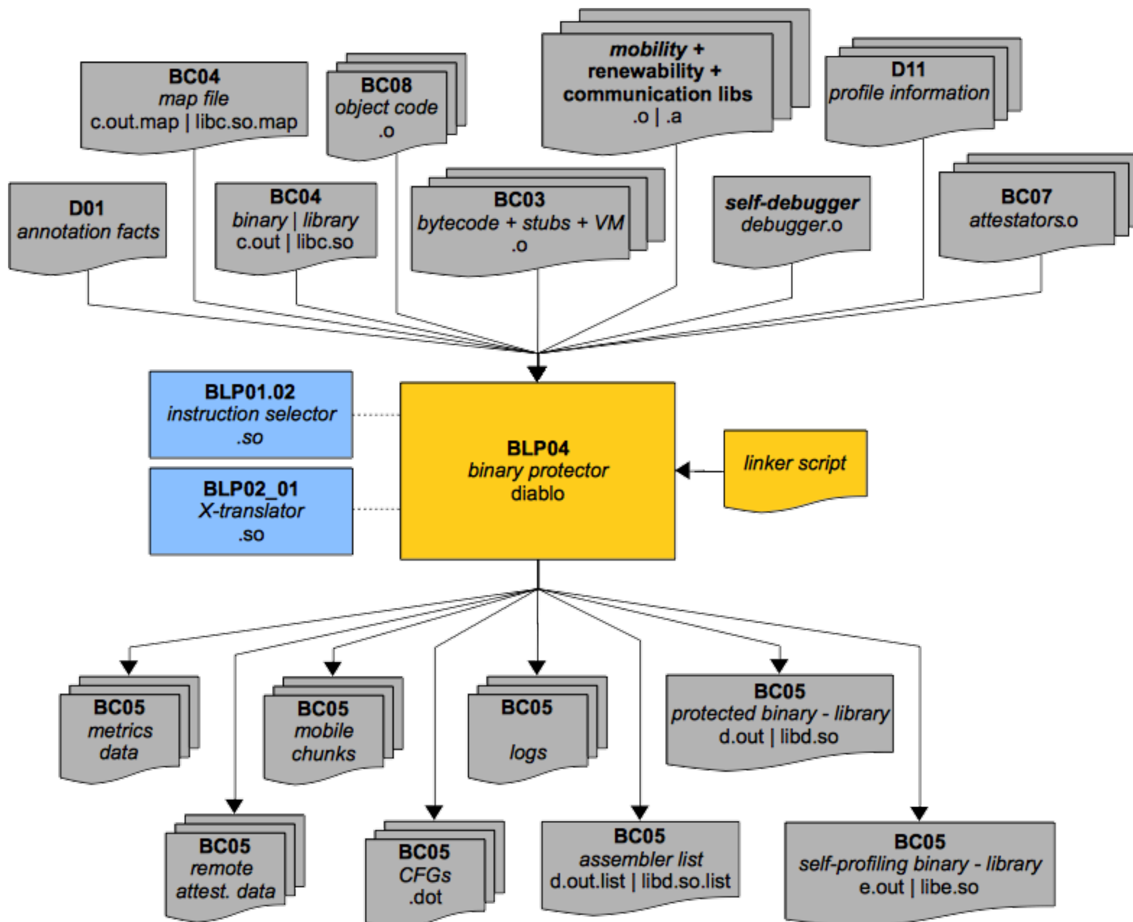

Figure 25: Integration of the SoftVM and application of binary-level protections.

8.6 BLP04 - Part 2: Binary-level Protections

In the same run of the Diablo tool BLP03, as depicted in Figure 25 a number of other binary protections are applied in step BLP04. The fragments and procedures to be protected are determined using the annotation facts in D01. The binary protections are implemented in two parts: transformations that are applied on the CFG representation Diablo has of the binary, and a variety of fix-ups that happen during code & data layout and assembly. UGent is responsible for all of the protections described except for Control Flow Tagging (which was developed and implemented by GTO).

The end results of this run are a protected application BC05, logs of the applied transformations, an annotated assembler listing, and metrics. We use the names d.out and libd.so in Figure 25 to mark that they correspond to rewritten versions of c.out and libc.so.

Transformations on the CFG

The first transformation to be applied is that of Control Flow Tagging (CFT), described in D2.08 and D3.09. After this, call-stack checks are inserted in the procedures to be protected (as described in D2.10). Code factoring (described in D2.06) is performed to couple the code from the original application and the code of the newly inserted components (such as the SoftVM) more tightly. Subsequently invocations of the reaction mechanisms are inserted all over the program (as described in D2.10 and D3.09), and then control flow obfuscations are applied. These obfuscations include the insertion of opaque predicates, the flattening of control flow, and the insertion of branch functions (which are all described in D2.06). The next transformation is that of self-debugging (described in D2.08 and D2.10), where parts of the code are transformed so that they can be executed in another context (that of the mini-debugger). Lastly there is the code mobility transformation (as described in D3.02), in which parts of the CFG are split off into separate CFGs.

After these transformations the CFG is laid out into long chain of instructions. This code layout happens in a randomized manner however, to ensure that the original code of the binary and newly inserted code are intertwined (this is described in D2.06).

Fix-ups During Layout and Assembly

Right before the process of code layout, some fix-ups for call-stack checks happen. After the code layout has been determined, the addresses for the regions that are to be attested, both locally (described in D2.10) and remotely (described in D3.06) are known. This thus allows us to generate the Attestation Data Structures (ADSs) and insert them in the binaries as data sections. The ADSs are also outputted as files together with the startup labels. This happens in the BC05 directory as `ads_ATTESTATOR_NAME` and `startup_labels_ATTESTATOR_NAME`, respectively. The data layout is subsequently determined, allowing some fix-ups for the SoftVM to proceed.

After layout has been finished for the main CFG, it can proceed for the other CFGs that were split off for code mobility. At this point some obfuscations and layout randomization can also be applied to allow for diversification of the resulting mobile blocks. These mobile blocks are outputted at this point together with some meta-data. The meta-data describes the associated data sections that were made mobile, and is not present if there are no such sections. These files are outputted either in BC05 or in an output directory given to BLP04 as an argument using the `-CMO` option. The ACTC chooses a date-dependent subdirectory of `BC05/mobile_blocks` as output directory. The filename for a mobile block is `mobile_dump_ID`, and for its meta-data is `mobile_dump_ID.metadata`. The ID is a hexadecimal number with a fixed width of 8 characters (e.g. 00003F22).

Finally the code of the protected binary is assembled into real instructions. This means we know the binary contents of the regions to be attested, and the checksums for these regions are thus calculated.

Composability

Currently a number of techniques do not compose and these combinations have thus been disabled. These are:

- Call-stack checks and code mobility: no call-stack checks will be inserted in locations that are slated to be made mobile
- Call-stack checks and self-debugging: no call-stack checks will be inserted in locations that are slated to be moved to the debugger context
- Control flow obfuscations and code mobility: no control flow obfuscations are applied in a region that is to be made mobile

- Control flow obfuscations and self-debugging: no control flow obfuscations are applied in a region that is to be moved to debugger context
- Code mobility and attestation (both local and remote): no attestation is performed on a region that is to be made mobile

Logging and Metrics

There are two forms of logging: some general Diablo logging and log files specific to each binary protection in which the successive transformation steps are described. All of these files are placed in the BC05 directory.

The general Diablo log has as name `diablo-obfuscator.log` and contains a lot of information about the workings of Diablo, a lot of which is unrelated to the binary protections that are applied. The verbosity of this log can be increased by adding the `-v` option (multiple times) as an argument to BLP04 in the ACTC config file. Also generated is an annotated assembler listing with the name `BINARY_NAME.list`. This file contains an entry for every instruction in the protected binary with the following information: the address of the instruction in the protected binary, the address of the instruction in the original binary, the opcode of the instruction, and the Diablo-phase in which the instruction was created. You can use the last one to find out which transformation was responsible creating a certain instruction.

The log files specific to a technique contain (amongst others) an entry for every transformation step. These entries contain a transformation ID and variety of information specific to the transformation such as its name, the address of a BBL, a function name, etc. To generate these entries the `--log-transformations` option should be passed to Diablo. These log files have a name specific to their technique such as `BINARY_NAME.diablo.obfuscation.log`. An example of a piece of such a log is:

```
751,OpaquePredicate,20848,acclExchange,'arm_opaque_predicate_3|(x^3-x)'
752,OpaquePredicate,20898,acclExchange,'arm_opaque_predicate_2|x^2div2'
753,OpaquePredicate,208a8,acclExchange,'arm_opaque_predicate_2|x+x'
```

These lines contain (in this case) a transformation ID, the name of the technique, the address of the BBL on which it is applied, the name of the function, the specific instance of the technique being applied.

If the `--dump-transformation` option is used as well, more verbose logs and dumps (such as dot graphs before and after the transformation) are generated for every transformation step in a subdirectory of BC05/transformation-logs that has as name the transformation ID. The path where these subdirectories are placed can be configured using the `--transformation-log-path` option. Specific log files are available for all protections that include transformations on the CFG, except for CFT, factoring, and reaction mechanisms.

Static and (if available) dynamic complexity metrics are generated for both the entire binary and the regions as described by the annotations. These are named as follows:

- `BINARY_NAME.stat_stat_complexity_info`;
- `BINARY_NAME.stat_dynamic_complexity_ino`;
- `BINARY_NAME.stat_stat_regions_complexity_info`;
- `BINARY_NAME.stat_dynamic_regions_complexity_info`

Individual protection techniques can also generate certain ad-hoc metrics. These files have a name specific to their technique such as `OUTPUT_NAME.code_mobility_metrics`.

Renewability

As described in deliverable D3.08 Section 3.1.1 renewability support has been added to diablo and the code mobility server allowing the creation of diversified mobile blocks.

Configuration

```
// Binary Code Control Flow Obfuscation
"BLP04": {
  "excluded"        : false,
  "options"         : ["--dump-transformations on",
               "--generate-dots-softvm",
               "--log-transformations on"],
  //Generate a self-profiling version of the obfuscated binary/library
  "self-profiling"  : false,
  //Use the runtime profiles generated in BLP00 when obfuscating
  "runtime_profiles": false,
  //Enable anti-debugging transformations
  "anti_debugging"  : true,
  //Enable binary obfuscation transformations
  "obfuscations"    : true,
  //Enable call-stack checks
  "call_stack_check": true,
  //Enable client side code-splitting using softVM
  "softvm"          : true,
  //Enable code mobility tranformations (mobile block generation)
  "code_mobility"   : true
},
```

9 Server-Side Deployment

source: deliverable d5.08 section 6, Deliverable D5.09 Section 5.5, extended
Section authors:
Alessandro Cabutto (UEL), Mariano Ceccato (FBK), Alessio Viticchié (POLITO), Jeroen Van Cleemput (UGent)

9.1 Deployment Scripts for Online Protection Techniques

The deployment of the server-side components of the online protection techniques have been automated using scripts provided by the tool developers. These scripts are called by the ACTC after compilation has finished. Script locations for client-server code splitting (P10), code mobility (P20) and remote attestation (P80) can be configured in the SERVER section of the ACTC configuration JSON.

9.2 Server side slice

When Client/Server code splitting is applied, part of the client code is removed from the application and it is moved to the server. Practically, this corresponds to generate a brand new file, called *slice.c*, to be deployed in the secure server.

The deployment script (option P10 in the SERVER section of aspire.json) has been provided to address this task. This script takes care of compiling the slice and of moving it into the secure server, where the slice manager is ready to run it as soon as a sliced client connects.

9.3 Server side RA components

At the end of the build phase, only the client side components of the RA infrastructure are generated (i.e., attestators). This task if managed by the RA deployment script, option P80 in the SERVER section of the ASPIRE configuration JSON file.

The script is invoked by the ACTC by passing the following parameters:

- the path of the RA annotations interpreter output folder, it is needed to understand how to characterize the Verifier and Extractor depending on each attestator that has been injected in the protected application and the attestation frequencies;
- the path of the BC05 output folder, it is needed to retrieve the ADS for the Extractor and to understand the `attest_at_startup` areas;
- the AID string of the just generated application.

Starting from these inputs, the deploy script is able to:

- compile the server side application dependent RA components, that are the verifiers and the extractors;
- deploy all the verifiers, extractors and ADSs in the proper place on the server;
- check the server status and sets up all the server environment: it ensures that RA manager and MySQL server are up and running;
- register the protected application and all its related data in the ASPIRE database.

9.4 Renewability Manager

Renewability Manager is the server-side component in charge of orchestrating delivery to client applications of diversified versions of specific mobile blocks. To achieve this goal it uses renewability support from UGent (see D5.08, Section 3.1) and relies on the Code Mobility Server (see D3.08, Section 1.1.1). A detailed overview of the manager can be found in deliverable D3.08 section 3.2. To support the renewability manager in the ACTC two new task have been added to the tool chain:

- task_SERVER_RENEWABILITY_CREATE : A wrapper task for the create_new_application.sh script, which registers the application in the renewability database of the server. The script takes one argument, the application ID (AID).
- task_SERVER_RENEWABILITY_POLICY: A wrapper task for the set_application_policy.sh script, which sets renewability policy for the application. The script has three arguments: the AID, the revision duration and a boolean indicating whether or not timeout is mandatory.

Both tasks are executed together with other server side deployment tasks after the ACTC has finished compiling the application. A renewability section has been added in the ACTC configuration.

9.5 Code Mobility Deployment

When Code Mobility is applied to a certain application, mobile code blocks are generated into BC05 of ACTC's build directory. A deployment script (pointed by P20 option of SERVER section in aspire.json file) has been provided in order to copy those blocks to a well known location in the file system. In fact the Code Mobility Server expects to find blocks into /opt/online_backends/APPLICATION_ID/code_mobility/REV_NUMBER. ACTC invokes the deployment script passing the application identifier as argument while the actual revision number is computed directly by the script. Finally the ASPIRE Portal is launched if no previous instances are found.

Configuration

```
// Server side management
"SERVER": {
  "excluded"   : false,
  "ip_address" : "YOUR_IP_HERE",

  // Code Splitting
  "P10": {
    "script": "/opt/client_server_splitter/server-deploy.sh"},

  // Code Mobility
  "P20": {
    "script": "/opt/code_mobility/deploy_application.sh"},

  // Remote Attestation
  "P80": {
    "script": "/opt/RA/deploy/deploy.sh"},

  // Renewability
  "RENEWABILITY": {
    "excluded"              : false,
    "new_application_script": "/opt/renewability/scripts/create_new_application.sh",
    "set_policy_script"     : "/opt/renewability/scripts/set_application_policy.sh",
    "revision_duration"     : "72000",
    "timeout_mandatory"     : false
  }
},
```

10 Metrics Generation and Collection

Source: Deliverable D5.08 Section 7, D5.09 Section Section 6, merged and updated to reflect latest release
Section authors:
Jeroen Van Cleemput, Bart Coppens (UGent)

The basic ACTC tool flow presented in the previous sections has been extended with several additional steps and output files for metrics generation and collection. These additions are shown in Figure 26. On the left side the original ACTC binary-level processing steps of the ACTC are shown greyed out and contained by a dotted grey line. The additional metrics-related steps are shown on the right side, contained by a dotted black line. In this section, we give a short overview of the different steps involved in generating and collecting metrics using the ACTC.

First, metrics are generated for the application version with only source code transformations applied. In case the ACTC does not perform any source code transformations, metrics are generated for the unprotected "vanilla" application. This is done by generating a self-profiling version of the vanilly binary, which generates an execution profile. The execution profile is generated by having the ACTC running the binary on a development board.

The execution profile can furthermore be used to steer the application of the binary protections to limit the execution overhead.

The input for this step is BC02, which is the output of the linker. The metrics are computed in three steps (BLP00_01_SP, BLP00_02_SP, and BLP00_03_SP where the SP stands for Self-Profiling):

1. **Generate self-profiling binaries (BLP00_01_SP)**

   - Input: object files (BC08), binary/library (BC02)
   - Output: self-profiling binary/library (BC02_SP)
   - Metrics: **.stat_complexity_info*

   In this first step the library or binary is rewritten to support the generation of run-time profiles. This step takes the binary or library from BC02 together with the object files from BC08 as input and generates a self-profiling binary or library in BC02 SP. The resulting binary is the same as BC02, except that the binary, when executed, tracks the execution counts of its basic blocks, which are dumped at the end of the execution.

   Static metrics are generated for the application protected with source code transformations, or the "vanilla" application in case not source code protections are applied.

2. **Run on target board (BLP00_02_SP)**

   - Input: self-profiling binary/library (BC02 SP)
   - Output: run-time profiles (BC02_SP/profiles)
   - Metrics: **.plaintext*

   Next, the self-profiling binaries are run on a target board to collect the run-time profiles. This step is automated using use-case specific scripts that copy the required files to the board and retrieve the run-time profiles to BC02_SP/profiles when the application has finished.

   To support source-only online protection techniques, such as client-server code splitting, the deploy scripts for those technique have to be called to make the code available on the server before running the application on the target board. To this end the server-side deployment script for client-server code splitting (task SERVER_P10) is now executed immediately after the server-side code has been generated in the SLP06 step.

3. **Recompile using run-time profile info (BLP00_02_DYN)**

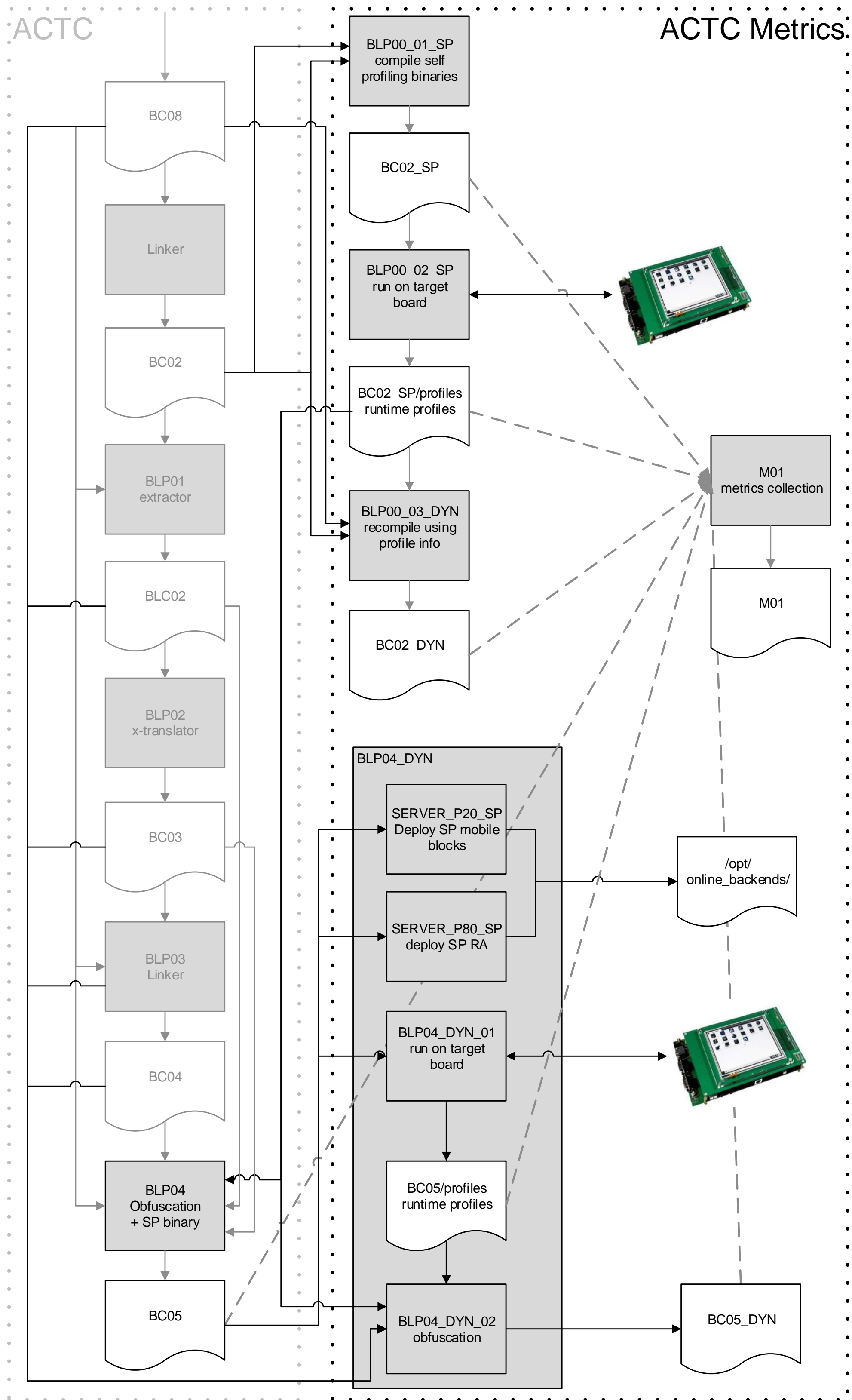

Figure 26: ACTC metrics subsystem supporting online protection techniques and dynamic metrics of obfuscated binaries

- Input: object files (BC08), binary/library (BC02), run-time profiles (BC02_SP/profiles)
- Output: binary/library compiled using run-time profiles (BC02_DYN)
- Metrics: **.dynamic_complexity_info*

Finally, the library or binary is rewritten based on the information contained in the run-time profiles and dynamic metrics are calculated.

To generate metrics for the applications protected with source code and binary protection techniques the diablo obfuscator has been extended to generate static metrics (BLP04) and dynamic metrics (BLP04_DYN). These dynamic metrics are collected on the final protected binary. This allows for more accurate metrics, compared to estimating the metrics based on propagating the profile information from BLP00_01 to the final protected binary.

1. **Diablo obfuscator (BLP04)**

   - Input: object files (BC08), binary/library (BC02), extractor output (BLC02), bytecode (BC03)
   - Output: protected binary/library and self-profiling binary/library(BC05)
   - Metrics: **.stat_complexity_info*, **.stat_regions_complexity_info*

   The diablo obfuscator has been extended to generate static metrics for both the complete binary or library and for the individual protected regions. The obfuscator has also been updated to generate both the protected binary/library and (optionally) a self-profiling version of the protected binary/library.

2. **Deployment of online techniques SERVER_P20_SP and SERVER_P80_SP** Since the self-profiling binaries differ from their non-self-profiling counterparts, both the remote attestation and code mobility techniques have to be deployed using the self-profiling binary when the self-profiling application is run on the target board in BLP04_DYN_01 task. To this end two new scripts, SERVER_P20_SP (code mobility) and SERVER_P80_SP (remote attestation) have been added to the ACTC as part of the BLP04_DYN task as shown in Figure 6. After the ACTC has finished compiling the application, the SERVER_P20 and SERVER_P80 tasks again deploy mobile blocks and remote attestators for the non-self-profiling application.

3. **Run on target board (BLP04_DYN_01)**

   - Input: obfuscated self-profiling binary/library (BC05)
   - Output: obfuscated binary run-time profiles (BC05/profiles)
   - Metrics: **.plaintext*

   Next, the obfuscated self-profiling binaries are run on a target board to collect the run-time profiles. This step is automated using use-case specific scripts that copy the required files to the board and retrieve the run-time profiles to BC5/profiles when the application has finished.

4. **Diablo obfuscator using run-time profile info (BLP04_DYN_02)**

   - Input: object files (BC08), binary/library (BC02), extractor output (BLC02), bytecode (BC03), run-time profiles (BC02_SP/profiles), obfuscated binary run-time profiles (BC05/profiles)
   - Output: None (BC05 DYN)
   - Metrics: **.dynamic_complexity_info*

   The diablo obfuscator is run again with the run-time profiles generated in BLP00_02_SP and BLP04_DYN_01 as additional inputs, generating dynamic metrics for the obfuscated binary.

Finally, the metrics collection step M01 collects the generated metrics file from each compilation step that produces metrics (indicated in black) and centralizes them in the M01 directory.

A detailed overview of the different metrics foreseen to be supported by this tool flow can be found in deliverable D4.06.

In addition to the metrics presented in that report, which are proposed as part of a general software protection evaluation methodology, but which are not all concrete enough yet and not broad enough yet to cover all our needs, we also implemented (late in the project) some more ad-hoc metrics to improve the support for measuring the overhead and the protection strength of attestation techniques (i.e., code guards and remote attestation) and of code mobility by having the ACTC produce the following new metrics:

- per region of code to be guarded/attested: the number of bytes stored in the Area Data Structure to describe the region
- per region of code to be guarded/attested: the number of blocks
- per region of code to be guarded/attested: the number of bytes to be guarded/attested
- per region to be made mobile: the number of mobile blocks
- per region to be made mobile: the total size of the mobile blocks

Configuration

```
// vanilla self-profiling
"BLP00": {
  "excluded": false,

  // generate vanilla self-profiling binary
  "_01": {
    "excluded": false,
    "options" : []
  },

  // collect execution profile on target board
  "_02": {
    "excluded": false,
    //External script to run the application and collect runtime profiles
    "script" : ""
  },

  // recompile using execution profile and calculate dynamic metrics
  "_03": {
    "excluded": false,
    "options" : []
  }
},

// Binary Code Control Flow Obfuscation
"BLP04": {
  "excluded"         : false,
  "options"          : ["--dump-transformations on",
               "--generate-dots-softvm",
               "--log-transformations on"],
  //Generate a self-profiling version of the obfuscated binary/library
  "self-profiling"   : false,
  //Use the runtime profiles generated in BLP00 when obfuscating
  "runtime_profiles": false,
  "anti_debugging"   : true,
  "obfuscations"     : true,
  "call_stack_check": true,
  "softvm"           : true,
  "code_mobility"    : true
},

// Generate dynamic metrics using diablo obfuscator
"BLP04_DYN": {
  "excluded": false,

  // collect execution profile on target board
  "_01": {
    "excluded": false,
    "options" : "",
    //External script to run the application and collect runtime profiles
    "script" : ""
  },

  // recompile using runtime profile and calculate dynamic metrics
  "_02": {
    "excluded": true,
    "options" : ""
  }
}
```

11 Caching ACTC

Section authors:
Jeroen Van Cleemput (UGent)

Source: Deliverable D5.09 Section 7, unmodified

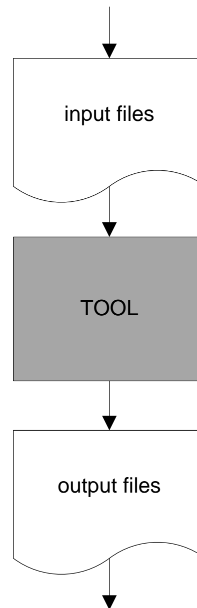

Figure 27: Fixed input and output folders in the non-caching ACTC

Since version 2.6.0, the ACTC has been extended with caching functionality. The caching mechanism significantly reduces the overhead of the ACTC when compiling different versions of the same application with different combinations of protection techniques applied. This is accomplished by saving intermediate results of the individual tools in a unique folder depending on the consumed annotations by the tools executed up until that point. To this end, two issues in the ACTC have been fixed: In the non-caching version of the ACTC, annotations are either applied during the SLP01 step, or are directly inserted into the source code. Any change in annotations therefore modifies the source code and triggers a complete recompilation of the application. A small parameter change in a remote attestation annotation for example, would trigger the unnecessary re-execution of all preceding tools in the chain. Furthermore, because of the fixed input and output folders of each tool, as shown in Figure 27, previously generated output of a tool is overwritten when the tool is re-executed. Intermediate results for different annotation combinations are therefore lost. To solve these two issues the following changes were made in the tool chain:

- Annotations are applied just before the tool that consumes them, eliminating the unnecessary execution of preceding tools when annotations for a specific tool are modified.
- Instead of a fixed input and output folder for each tool, the input and output folders of the tools now depend on (a hash of) the specific annotations consumed by the tools executed up until that point.

Figure 28 illustrates these two changes: First, the original source files are copied to a new input folder and annotated. The name of this new folder is a combination of the original folder name, together with a postfix tag derived from both the annotations consumed by the tool as well as all the annotations consumed by the preceding tools in the chain. Each combination of annotations

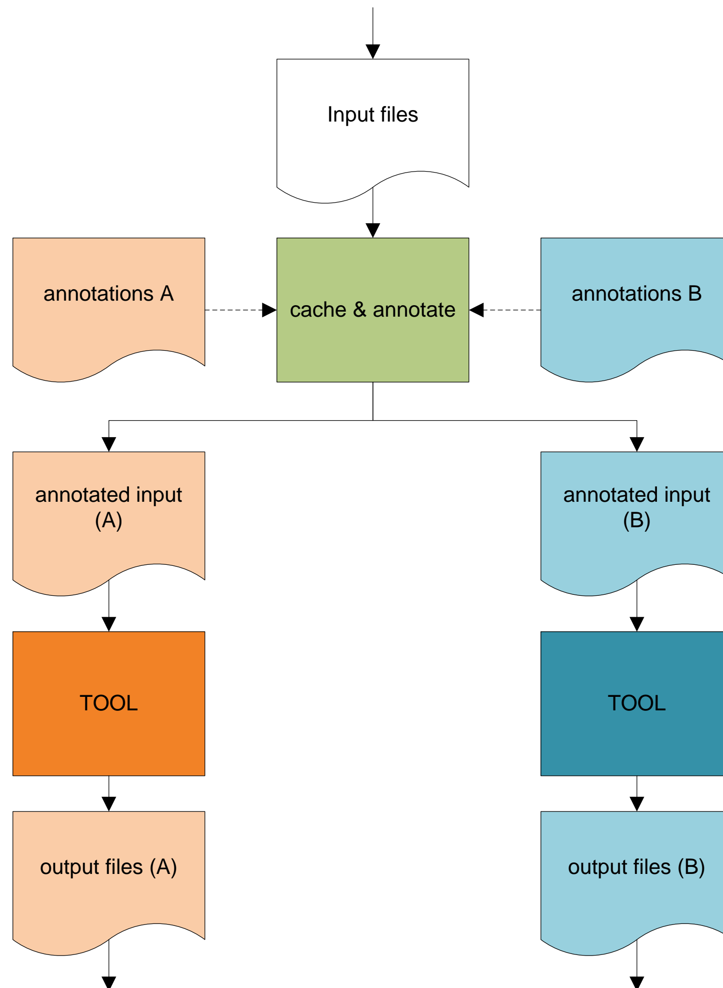

Figure 28: Caching and annotation rewriting flow

applied up until this point will therefore result in a unique input folder name. When annotations A (orange) are applied, the original input files are copied to the "annotated input (A)" folder. Likewise when annotations B (blue) are applied, the original input files are copied to the "annotated input (B)" folder. Only annotations relevant to the tool in question are applied. Next, the tool is executed on the contents of the new annotated input folder. The output folder of the tool consists of the fixed output folder of that specific tool, combined with the same postfix of the input folder. Each combination of annotations applied up until this point will therefore also result in a unique input folder name. The output folders are then used as input to the next tool in the chain. Caching and annotation rewriting is done before the execution of all source-level transformation tools. To support caching of the binary transformation tools in Diablo, all relevant annotations for the binary tools are inserted just before the annotation extraction tool SLP04 is run. Finally, after compilation is finished, symbolic links are created in the build folder, linking the original fixed output names (as used in the non-caching ACTC) to the postfixed folders of the latest build.

12 License Tool Example

Section authors:
Jeroen Van Cleemput, Bart Coppens (UGent)

Source: Deliverable 5.06 v1.2 Section 6, updated to reflect latest release

This section is intended as a walk-through to show users how to set up the ACTC and compile a simple "hello world"-like application, protected with a set of protection techniques. We give a short overview of the application and how it is annotated; explain the general usage of the ACTC and how to set up the ACTC to compile a specific application. Finally, we discuss the compilation process by analysis the ACTC output.

12.1 License Example Description

In this section, we discuss the ACTC work-flow using a small hello-world like application called 'license'. To keep the compilation output as small as possible and keep the complexity of the compilation process to a minimum this tool consists of a single C file containing a single main function together with additional library code that remains unprotected.

```
-rw-r--r-- 1 aspire aspire  731 Apr 12 17:43 library.c
-rw-r--r-- 1 aspire aspire   26 Apr 12 17:43 library.h
-rw-r--r-- 1 aspire aspire 2359 Apr 13 14:14 license.c
```

The example can be found in the framework repository in the following directory:

/framework/testing/ACTC/examples/license/

The functionality of the program is very limited: It reads a license number from the command line and it checks if the license is still valid (a license lasts 30 days). The license number, in fact, contains the information about the license activation date.

The license number is in the form "DD MM YYYY", it represents the activation date. If the license is still valid the program outputs "Yes". If the license is expired, the program outputs "No".

For development/testing purposes, the program also outputs the difference between the activation date and the current date. We opted for this example for several reasons:

- It offers close to the least amount of complexity while still allowing demonstration of some annotations and the functioning of the tool chain.
- This example is used as a unit test by all project partners before they release new protection plug-ins.

12.2 Source Code Annotations

To protect the license application, annotations have been added to the license.c file for data obfuscation,binary code obfuscations, anti-debugging, code guards and client-server code splitting. The annotated source code is shown below:

```
#include
#include
#include
#include
<stdio.h>
```

```
<stdlib.h>
<time.h>
"library.h"

#define DOBFS __attribute__((ASPIRE("protection(xor,mask(constant(35)))")))
int
main (int argc, char **argv)
{
    _Pragma ("ASPIRE begin protection (obfuscations,enable_obfuscation(
      branch_function:percent_apply=10,opaque_predicate:percent_apply=10))")

    int day1 DOBFS = 0;
    int mon1 DOBFS = 0;
    int year1 DOBFS = 0;
    int day2 DOBFS ;
    int mon2 DOBFS ;
    int year2 DOBFS ;
    int ref DOBFS ;
    int dd1 DOBFS ;
    int dd2 DOBFS ;
    int delta DOBFS ;
    int i;
    time_t t = time (0);
    struct tm tm = *localtime (&t);


    _Pragma ("ASPIRE begin protection(anti_debugging)")
    for (i = 0; argv1i != '\0'; i++)
    {
        day1 = 10 * day1 + (char) argv1i - '0';
    }
    for (i = 0; argv2i != '\0'; i++)
    {
        mon1 = 10 * mon1 + (char) argv2i - '0';
    }
    for (i = 0; i < 4; i++)
    {
        year1 = 10 * year1 + (char) argv3i - '0';
    }
    _Pragma ("ASPIRE end");


    _Pragma("ASPIRE begin protection(guarded_region,label(GR1))")
    day2 = tm.tm_mday;
    mon2 = tm.tm_mon + 1;
    year2 = tm.tm_year + 1900;
    ref = year1;
    dd1 = 0;
    dd1 = days_at_month (mon1);
    for (i = ref; i < year1; i++)
    {
        if (i % 4 == 0)
        {
            dd1 += 1;
        }
    }
    dd1 = dd1 + day1 + (year1 - ref) * 365;
    dd2 = 0;
    for (i = ref; i < year2; i++)
    {
        if (i % 4 == 0)
        {
            dd2 += 1;
        }
    }
    _Pragma ("ASPIRE end");


    dd2 = days_at_month (mon2) + dd2 + day2 + ((year2 - ref) * 365);
    delta = dd2 - dd1;


    _Pragma ("ASPIRE begin protection (barrier_slicing, criterion(dd1), label(slicing))")
    if ((0 <= delta) && (delta <= 30))
```

```
    {
    printf ("Yes %d\n", dd2 - dd1);
    }
    else
    {
    printf ("No %d\n", dd2 - dd1);
    }
    _Pragma ("ASPIRE end")

    return 0;

    _Pragma ("ASPIRE end")
}
```

12.3 ACTC Usage

Running the ACTC with the flag shows the different command-line options the tool supports. In all tool outputs and logs we present in the remainder of this section, additional **comments** are inserted in red to explain the content of the outputs and logs. Additionally, we sometimes marked text as **bold** and/or **underlined** to mark different sections of the outputs and logs.

```
usage: actc.py [-h] [--version] [-j N] [-d] [-p] [-v] [-a] [-f configName]
               [-g [configName]] [-u [configName]]
               [build,clean]

ASPIRE Compiler Tool Chain

positional arguments:
  build,clean           ACTC commands [build]

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit

Build:
  -j N, --jobs N        allow 1..N jobs at once [1]
        The number of simultaneous tasks/jobs that can be executed by
        the DoIt build system.
  -d, --debug           print debugging informations
        Let the ACTC and the tools used by the ACTC produce debug
        information.
  -p, --process         generate processing graph
        The ACTC generates a graph representing the complete
        compilation process from start to finish.
  -v, --verbose         print everything from a task
        Prints additional task information such as the exact command
        to run each tool.
  -a, --aid             only generate the application id (AID)
        Only generate an application ID, do not start the compilation
        process.
Configuration:
  -f configName, --file configName
                        read configName [aspire.json]
        The ACTC configuration file used to compile the application.
        If no filename is provided, the ACTC tries to load the
        aspire.json file.
  -g [configName], --generate [configName]
                     generate a template configuration file [aspire.json]
        Generate a new (template) ACTC configuration file using
        default values.
        Used when starting a new project.
  -u [configName], --update [configName]
                        update old configuration file [aspire.json]
        Update the ACTC configuration file to the latest version.
        The ACTC will not accept a configuration file with an
        incompatible version number


ACTC v 2.8.0 ACTC version number
```

12.4 ACTC Configuration JSON File

An ACTC configuration file for for the license example is shown below. Sections unimportant for this application have been grayed out.

```
// ACTC 2.8.0
//
// Note:
// - "excluded": true/false [false]
//    if true, step is excluded from tool chain --> no output folder is
//    created
//    use this field to start tool chain from any step
//
// - "traverse": true/false [false]
//    if true, input files are copied to output folder without any change
//
{

  // Target platforms:
  // - Linux [default]
  // - Android
  "platform" :                  "linux",

  // Tools
  "tools": {
    // libraries
    "third_party":              "/opt/3rd_party",
    // src2src
    "annotation_reader":        ["perl",
                                  "/opt/wbc/annotation_reader.prl"],
    "config":                   "/opt/wbc/config.x",
    "wbta":                     ["python",
                                  "/opt/wbc/wbta/Wbta.py"],
    "convert_pragmas":          ["python",
                                  "/opt/wbc/convert_pragmas.py"],
    "wbc":                      "/opt/wbc/wbc.x",
    "read_annot":               "/opt/annotation_extractor/readAnnot.sh",
    "data_obfuscate":           "/opt/data_obfuscator/scripts/data_obfuscate.sh",
    "client_server_splitter": "/opt/client_server_splitter",
    "csurf":                    "/opt/codesurfer/csurf/bin/csurf",
    "codeguard":                "/opt/codeguard/codeguard.py",
    "anti_cloning":             "/opt/anti_cloning/annotation/replace.sh",
    "attestator_selector":      "/opt/RA/attestator_selector.sh",
    "reaction_unit":            "/opt/reaction_unit/script/replace.sh",
    "dcl":                      "/opt/dcl",
    "cft":                      "/opt/cf_tagging/cf_tagging.py",
    // src2bin
    "frontend":                 "/opt/diablo-toolchains/llvm3.4/bin/clang",
    // bin2bin
    "extractor":                "/opt/diablo/bin/diablo-extractor",
    "xtranslator":              "/opt/xtranslator/xtranslator",
    "code_mobility":            "/opt/code_mobility",
    "accl":                     "/opt/ACCL",
    "ascl":                     "/opt/ASCL",
    "anti_debugging":           "/opt/anti_debugging",
    "obfuscator":               "/opt/diablo/bin/diablo-obfuscator",
    "obfuscator_sp":            "/opt/diablo/bin/diablo-selfprofiling",
    "renewability":             "/opt/renewability"
  },

  // Source-level Tool chain
  "src2src": {
    "excluded": false,

    // Source code annotation
    "SLP01": {
      "excluded":                 false,
      "traverse":                 false,
      "annotations_patch":        "",
      "external_annotations":     "",
      "source"  :                 ["src/*.c",
                    "src/*.h"]
    },
```

```
// white-box crypto
"SLP03": {
  "excluded": false,
  "traverse": true,
  "renewability_script": true,
  // WBC seed (random, aid, none)
  "seed": "none",

  // WBC annotation extraction tool
  "_01": {
    "excluded": false
  },

  // White-Box Tool python
  "_02": {
    "excluded": false
  },

  // WBC header inclusion
  "_03": {
    "excluded": false
  },

  // preprocessor
  "_04": {
    "excluded": false
  },

  // WBC source rewriting tool
  "_05": {
    "excluded": false,
    "options":  ["-size 2000MB"]
  }
},

// preprocessor
"SLP02": {
  "excluded": false
},

// data hiding
"SLP05": {
  "excluded": false,
  "traverse": false,

  // source code analysis
  "_01": {
    "excluded": false,
    "options" : []
  },

  // data obfuscation
  "_02": {
    "excluded": false,
    "options" : []
  }
},

// client server clode splitting
"SLP06": {
  "excluded": false,
  "traverse": false,

  // Process
  "_01": {
    "excluded": false,
    "options" : ["-i"]
  },

  // CSurf
  "_02": {
    "excluded": false
  },

  // Code transformation
```

```
      "_03": {
        "excluded": false
      }
    },

    // annotation extraction + external annotation file(s)
    "SLP04": {
      "excluded": false,
      "options" : [],
      "external": []
    },

    // code guard
    "SLP08": {
      "excluded": false,
      "traverse": false,
      "options" : []
    },

    // anti-cloning
    "SLP09": {
      "excluded": false,
      "traverse": true,
      "options" : []
    },

    // remote attestation
    "SLP07": {
      "excluded": true,
      "options" : []
    },

    // reaction unit
    "SLP10": {
      "excluded": false,
      "traverse": true,
      "options" : []
    },

    // diversified crypto library
    // only applicable for ANDROID platform
    "SLP11": {
      "excluded": false,
      "traverse": true,
      "options" : []
    },

    // control flow tagging
    "SLP12": {
      "excluded": false,
      "traverse": true,
      "options" : []
    }
  },

  // Assembler, Compiler, Linker
  "src2bin": {
  "excluded": false,
    // Common options for all tools
    "options"       : ["-ccc-gcc-name   arm-diablo-linux-gnueabi",
                       "-gcc-toolchain /opt/diablo-gcc-toolchain",
                       "-isysroot      /opt/diablo-gcc-toolchain/"
                       + "arm-diablo-linux-gnueabi/sysroot",
                       "-target        arm-diablo-linux-gnueabi"],

    "PREPROCESS": {
      // -I <dir>
      // -isystem <dir>
      // -include <file>
      // -D<macro[=defn]>
      "options"     : []
    },

    // .c, .cpp
    "COMPILE": {
```

```
      "options"    : ["-mcpu=cortex-a8",
                      "-no-integrated-as"],
      "options_c"  : [],
      "options_cpp": []
    },

    // accl.c
    "COMPILE_ACCL": {
      "protocol"    : "http",
      "endpoint"    : "127.0.0.1",
      "port"        : "8088",
      "file_path"   : ""
    },

    // Linker
    "LINK": {
      "options"    : ["-Wl,--no-demangle",
                      "-Wl,--no-merge-exidx-entries",
                      "-marm"],
      // basename of linked file
      //   if empty, default value computed from options:
      //   "liba.so" if "-shared" else "a.out"
      "binary"     : ""
    }
  },

  // Binary Rewriting Tool Chain
  "bin2bin": {
    "excluded": false,

    // bytecode diversity seed, integer or RANDOM
    "bytecode_diversity_seed"     : "0",
    "code_mobility_diversity_seed": "0",

    // vanilla self-profiling
    "BLP00": {
      "excluded": true,


      // generate vanilla self-profiling binary
      "_01": {
        "excluded": false,
        "options" : []
      },

      // collect execution profile on target board
      "_02": {
        "excluded": false,
        "script" : ""
      },

      // recompile using execution profile and calculate dynamic metrics
      "_03": {
        "excluded": false,
        "options" : []
      }
    },

    // Native Code Extraction
    "BLP01": {
      "excluded": false,
      "traverse": false,
      "options" : []
    },

    // Bytecode Generation
    "BLP02": {
      "excluded": false,
      "options" : []
    },

    // Code Integration
    "BLP03": {
      "excluded": false,
      "options" : []
```

```
    },

    // Binary Code Control Flow Obfuscation
    "BLP04": {
      "excluded"        : false,
      "options"         : [],
      "self-profiling"  : false,
      "runtime_profiles": false,
      "anti_debugging"  : true,
      "obfuscations"    : true,
      "call_stack_check": true,
      "softvm"          : true,
      "code_mobility"   : true
    },

    // Generate dynamic metrics using diablo obfuscator
    "BLP04_DYN": {
      "excluded": true,

      // collect execution profile on target board
      "_01": {
        "excluded": false,
        "options" : "",
        "script" : ""
      },

      // recompile using execution profile and calculate dynamic metrics
      "_02": {
        "excluded": true,
        "options" : ""
      }
    }
  },

  // Server side management
  "SERVER": {
    "excluded"   : true,
    "ip_address" : "",

    // Code Splitting
    "P10": {
      "script": ""},

    // Code Mobility
    "P20": {
      "script": ""},

    // Remote Attestation
    "P80": {
      "script": ""},

    // Renewability
    "RENEWABILITY": {
      "excluded"              : false,
      "new_application_script":
         "/opt/renewability/scripts/create_new_application.sh",
      "set_policy_script"     :
          "/opt/renewability/scripts/set_application_policy.sh",
      "revision_duration"     : "72000",
      "timeout_mandatory"     : false
    }
  },

  // Metric collection
  "METRICS": {
    "excluded" : false,
    "files"    : {
        "BC02_SP"          : ["*.stat_complexity_info"],
        "BC02_SP/profiles" : ["*.plaintext"],
        "BC02_DYN"         : ["*.dynamic_complexity_info"],
        "BC05"             : ["*.stat_complexity_info",
                             "*.stat_regions_complexity_info"],
        "BC05/profiles"    : ["*.plaintext"],
        "BC05_DYN"         : ["*.dynamic_complexity_info"]
    }
```

```
  },

  // Post-processing
  "POST": {
    // Short description in ACTC trace
    "brief": "",
    // Command line arguments
    "args" : ""
  }
}
```

12.5 Setting the Correct Tool Versions

Between major releases of the framework it is important to keep track of which versions of the tools are compatible with each other and with the latest version of the ACTC.
At the moment of writing, the following tools in the /opt directory are used:

```
3rd_party              -> /home/aspire/framework/testing/3rd_party
ACCL                   -> /home/aspire/framework/testing/ACCL
ACTC                   -> /home/aspire/framework/development/ACTC/main/src/
android-ndk            -> /etc/alternatives/android-ndk
android-sdk            -> /etc/alternatives/android-sdk
annotation_extractor   -> /home/aspire/framework/opt/annotation_extractor
anti_cloning           -> /home/aspire/framework/development/anti_cloning
anti_debugging         -> /home/aspire/framework/testing/anti_debugging
ASCL                   -> /home/aspire/framework/testing/ASCL
cf_tagging             -> /home/aspire/framework/development/cf_tagging
client_server_splitter -> /home/aspire/framework/testing/client_server_splitter/intraprocedural/
codeguard              -> /home/aspire/framework/development/codeguard
code_mobility          -> /home/aspire/framework/testing/code_mobility
codesurfer
data_obfuscator        -> /home/aspire/framework/development/data_obfuscator/tags/3.0.1
dcl                    -> /home/aspire/framework/development/dcl
diablo                 -> /home/aspire/framework/testing/diablo
diablo-android-gcc-toolchain  -> /etc/alternatives/diablo-android-gcc-toolchain
diablo-android-llvm-toolchain -> /etc/alternatives/diablo-android-llvm-toolchain
diablo-gcc-toolchain          -> /etc/alternatives/diablo-gcc-toolchain
diablo-llvm-toolchain         -> /etc/alternatives/diablo-llvm-toolchain
diablo-toolchains
home-software
online_backends
RA                     -> /home/aspire/framework/testing/RA
reaction_unit          -> /home/aspire/framework/development/reaction_unit
renewability           -> /home/aspire/framework/development/renewability
txl
wbc                    -> /home/aspire/framework/opt/wbc
xtranslator            -> /home/aspire/framework/testing/xtranslator
```

Switching to the latest set of tools can always be done using a script, which is kept up to date whenever tool versions change:

```
~/framework/vm/set-tool-versions.sh
```

Individual tools can be configured using the following script:

```
aspire@vm:~/framework$ ./select-tool-version.sh
```

12.6 Compiling the License Example

After configuring the ACTC and setting up the tool versions, the application is compiled by running the following command. The -p flag tells the ACTC to generate a graph representing the

compilation process.

```
aspire@vm:~/license$ /opt/ACTC/actc.py -p
```

The command-line output of the compilation process is shown below. For clarity, the different compilation steps are indicated by using an alternating white-light grey color scheme. Remarks about each compilation step are added in red.

Start of source to source transformations
SLP01: get source code (with annotations) → SC02

```
. SLP01
  create          /home/aspire/license/build/SC02
. SLP01
  copy            /home/aspire/license/src/library.c
  into            /home/aspire/license/build/SC02/library.c
. SLP01
  copy            /home/aspire/license/src/license.c
  into            /home/aspire/license/build/SC02/license.c
. SLP01
  copy            /home/aspire/license/src/library.h
  into            /home/aspire/license/build/SC02/library.h
```

SC02 → split C files → SC03

```
. SPLIT_C
  create          /home/aspire/license/build/SC03
. SPLIT_C
  copy            /home/aspire/license/build/SC02/library.c
  into            /home/aspire/license/build/SC03/library.c
. SPLIT_C
  copy            /home/aspire/license/build/SC02/license.c
  into            /home/aspire/license/build/SC03/license.c
. SPLIT_C
  copy            /home/aspire/license/build/SC02/library.h
  into            /home/aspire/license/build/SC03/library.h
```

SC02 → split .cpp files → SC09
No source code transformations are performed on .cpp files, they are copied the the output folder of the last source code transformation SC09.

```
  create          /home/aspire/license/build/SC12
. SPLIT_CPP
  copy            /home/aspire/license/build/SC02/library.h
  into            /home/aspire/license/build/SC12/library.h
```

SLP03: SC03 → white-box crypto → SC04 (Traversed)
White-box crypto is configured to be traversed in the ACTC configuration file for this application. No transformations are applied in this step, source files are copied directly to the destination folder

```
  create          /home/aspire/license/build/SC04
. SLP03
  copy            /home/aspire/license/build/SC03/library.h
  into            /home/aspire/license/build/SC04/library.h
. SLP03
  copy            /home/aspire/license/build/SC03/library.c
  into            /home/aspire/license/build/SC04/library.c
. SLP03
  copy            /home/aspire/license/build/SC03/license.c
  into            /home/aspire/license/build/SC04/license.c
```

SLP02: SC04 → preprocessor → SC05
Source code files are pre-processed using the front-end tool configured in the ACTC configuration file.

```
  create          /home/aspire/license/build/SC05
. SLP02_PREPROCESS
  preprocess      /home/aspire/license/build/SC04/library.c
  into            /home/aspire/license/build/SC05/library.c.i
. SLP02_PREPROCESS
  preprocess      /home/aspire/license/build/SC04/license.c
  into            /home/aspire/license/build/SC05/license.c.i
```

SLP05: SC05 → data obfuscation → SC06

```
  create          /home/aspire/license/build/SC05/log
. SLP05_02_OBFUSCATE
  obfuscate data /home/aspire/license/build/SC05/license.c.i
  into           /home/aspire/license/build/SC05/license.c.i.obf
INFO: using seed 0
. SLP05_02_OBFUSCATE
  obfuscate data /home/aspire/license/build/SC05/library.c.i
  into           /home/aspire/license/build/SC05/library.c.i.obf
. SLP05_02_COPY
  create          /home/aspire/license/build/SC06
. SLP05_02_COPY
  copy           /home/aspire/license/build/SC05/library.c.i.obf
  into           /home/aspire/license/build/SC06/library.c.i
. SLP05_02_COPY
  copy           /home/aspire/license/build/SC05/license.c.i.obf
  into           /home/aspire/license/build/SC06/license.c.i
```

SLP06: SC06 → client server code splitting → SC07

```
. SLP06_01_PROCESS
  create          /home/aspire/license/build/SC06.01/facts
. SLP06_01_PROCESS
  create          /home/aspire/license/build/SC06.01
. SLP06_01_PROCESS
  process         /home/aspire/license/build/SC06/license.c.i
  into            /home/aspire/license/build/SC06.01/license.c.i
16/10/25 16:37:46 - STEP1: converting pragmas to ASPIRE format
16/10/25 16:37:46 - STEP1: /opt/client_server_splitter/scripts/pragma_conv
ersion.py /home/aspire/license/build/SC06/license.c.i /home/aspire/license
/build/SC06/license.c.i.4534.tmp.2
16/10/25 16:37:46 - STEP2: applying normalization on file /home/aspire/lic
ense/build/SC06/license.c.i.4534.tmp.2
16/10/25 16:37:46 - STEP2: /opt/client_server_splitter/scripts/txl/norm.x
-s 1000 -o /home/aspire/license/build/SC06/license.c.i.4534.tmp.2 /home/as
pire/license/build/SC06/license.c.i.4534.tmp.2
16/10/25 16:37:46 - STEP3: checking for splitting annotations in /home/asp
ire/license/build/SC06/license.c.i.4534.tmp.2
16/10/25 16:37:46 - STEP3: /opt/client_server_splitter/check_annotations.sh
/home/aspire/license/build/SC06/license.c.i.4534.tmp.2
16/10/25 16:37:46 - STEP3.a: /opt/client_server_splitter/scripts/txl/check
_annotations.x -s 1000 /home/aspire/license/build/SC06/license.c.i.4534.tmp.2
16/10/25 16:37:46 - STEP4: extracting splitting annotations from file /hom
e/aspire/license/build/SC06/license.c.i.4534.tmp.2
16/10/25 16:37:46 - STEP4: /opt/client_server_splitter/scripts/txl/annotat
ions.x -s 1000 -o /home/aspire/license/build/SC06/license.c.i.4534.tmp.3
/home/aspire/license/build/SC06/license.c.i.4534.tmp.2 - -homedir
/home/aspire/license/build/SC06.01/facts/ -filename license.c.i
16/10/25 16:37:47 - STEP5: converting ASPIRE pragmas to original format
16/10/25 16:37:47 - STEP5: /opt/client_server_splitter/scripts/pragma_conv
ersion.py -r /home/aspire/license/build/SC06/license.c.i.4534.tmp.3
/home/aspire/license/build/SC06.01/license.c.i
. SLP06_01_PROCESS
  process         /home/aspire/license/build/SC06/library.c.i
  into            /home/aspire/license/build/SC06.01/library.c.i
16/10/25 16:37:47 - STEP1: converting pragmas to ASPIRE format
16/10/25 16:37:47 - STEP1: /opt/client_server_splitter/scripts/pragma_conversion.py /home/aspire/
    license/build/SC06/library.c.i /home/aspire/license/build/SC06/library.c.i.4563.tmp.2
16/10/25 16:37:47 - STEP2: applying normalization on file /home/aspire/license/build/SC06/library
    .c.i.4563.tmp.2
16/10/25 16:37:47 - STEP2: /opt/client_server_splitter/scripts/txl/norm.x -s 1000 -o /home/aspire
    /license/build/SC06/library.c.i.4563.tmp.2 /home/aspire/license/build/SC06/library.c.i.4563.
    tmp.2
16/10/25 16:37:47 - STEP3: checking for splitting annotations in /home/aspire/license/build/SC06/
    library.c.i.4563.tmp.2
16/10/25 16:37:47 - STEP3: /opt/client_server_splitter/check_annotations.sh /home/aspire/license/
    build/SC06/library.c.i.4563.tmp.2
16/10/25 16:37:47 - STEP3.a: /opt/client_server_splitter/scripts/txl/check_annotations.x -s 1000
    /home/aspire/license/build/SC06/library.c.i.4563.tmp.2
16/10/25 16:37:47 - STEP3: no client/server splitting annotation found, processing not required.
16/10/25 16:37:47 - STEP3: copying /home/aspire/license/build/SC06/library.c.i to /home/aspire/
    license/build/SC06.01/library.c.i
. SLP06_02_CSURF
  create          /home/aspire/license/build/SC06.01/csurf-project
. SLP06_02_CSURF
  csurf init      /home/aspire/license/build/SC06.01/license.c.i
                  /home/aspire/license/build/SC06.01/library.c.i
```

```
  into          /home/aspire/license/build/SC06.01/csurf-project/project.prj
csurf: Logging to project.prj_files/log.txt...
/home/aspire/license/build/SC06.01/license.c.i:165:12: warning: conflicting types for built-in
    function 'snprintf' [enabled by default]
/home/aspire/license/build/SC06.01/license.c.i:166:12: warning: conflicting types for built-in
    function 'vsnprintf' [enabled by default]
/home/aspire/license/build/SC06.01/license.c.i:203:15: warning: conflicting types for built-in
    function 'fwrite' [enabled by default]
/home/aspire/license/build/SC06.01/license.c.i:239:14: warning: conflicting types for built-in
    function 'malloc' [enabled by default]
/home/aspire/license/build/SC06.01/license.c.i:240:14: warning: conflicting types for built-in
    function 'calloc' [enabled by default]
/home/aspire/license/build/SC06.01/license.c.i:241:14: warning: conflicting types for built-in
    function 'realloc' [enabled by default]
/home/aspire/license/build/SC06.01/license.c.i:282:15: warning: conflicting types for built-in
    function 'strftime' [enabled by default]
/home/aspire/license/build/SC06.01/license.c.i: In function 'main':
/home/aspire/license/build/SC06.01/license.c.i:293:0: warning: ignoring #pragma ASPIRE begin [-
    Wunknown-pragmas]
/home/aspire/license/build/SC06.01/license.c.i:342:0: warning: ignoring #pragma ASPIRE begin [-
    Wunknown-pragmas]
/home/aspire/license/build/SC06.01/license.c.i:344:0: warning: ignoring #pragma ASPIRE end [-
    Wunknown-pragmas]
/home/aspire/license/build/SC06.01/license.c.i:350:0: warning: ignoring #pragma ASPIRE end [-
    Wunknown-pragmas]
/home/aspire/license/build/SC06.01/library.c.i:368:12: warning: conflicting types for built-in
    function 'snprintf' [enabled by default]
/home/aspire/license/build/SC06.01/library.c.i:372:12: warning: conflicting types for built-in
    function 'vsnprintf' [enabled by default]
/home/aspire/license/build/SC06.01/library.c.i:464:15: warning: conflicting types for built-in
    function 'fwrite' [enabled by default]
/home/aspire/license/build/SC06.01/library.c.i:584:14: warning: conflicting types for built-in
    function 'malloc' [enabled by default]
/home/aspire/license/build/SC06.01/library.c.i:586:14: warning: conflicting types for built-in
    function 'calloc' [enabled by default]
/home/aspire/license/build/SC06.01/library.c.i:588:14: warning: conflicting types for built-in
    function 'realloc' [enabled by default]
csurf: Building project.prj...
. SLP06_03_PREPROCESS_CLIENT
  create        /home/aspire/license/build/SC07
. SLP06_03_PREPROCESS_CLIENT
  preprocess    /opt/client_server_splitter/libraries/client/accl-message-wrapper.c
  into          /home/aspire/license/build/SC07/accl-message-wrapper.c.i
. SLP06_03_TRANSFORMATION
  create        /home/aspire/license/build/SC07/log
. SLP06_03_TRANSFORMATION
  create        /home/aspire/license/build/SCS01
. SLP06_03_TRANSFORMATION
  transform     /home/aspire/license/build/SC06.01/license.c.i
  into          /home/aspire/license/build/SC07/license.c.i
16/10/25 16:37:48 - STEP7: running codesurfer analysis on /home/aspire/license/build/SC06.01/
    license.c.i
16/10/25 16:37:48 - STEP7: /opt/client_server_splitter/codesurfer.sh /home/aspire/license/build/
    SC06.01/license.c.i /home/aspire/license/build/SC06.01/facts/ /home/aspire/license/build/SC06
    .01/csurf-project/
16/10/25 16:37:48 - STEP7.a: extracting defs, uses and defuses from /home/aspire/license/build/
    SC06.01/license.c.i
16/10/25 16:37:48 - STEP7.a: csurf -nogui -l /opt/client_server_splitter/scripts/codesurfer/du-
    script.stk /home/aspire/license/build/SC06.01/csurf-project//project -args /home/aspire/
    license/build/SC06.01/license.c.i
16/10/25 16:37:48 - STEP7.b: extracting types from /home/aspire/license/build/SC06.01/license.c.i
16/10/25 16:37:48 - STEP7.b: csurf -nogui -l /opt/client_server_splitter/scripts/codesurfer/
    functions.stk /home/aspire/license/build/SC06.01/csurf-project//project -args /home/aspire/
    license/build/SC06.01/license.c.i
16/10/25 16:37:49 - STEP7.c: extracting barrier slice from /home/aspire/license/build/SC06.01/
    license.c.i
16/10/25 16:37:49 - STEP7.c: csurf -nogui -l /opt/client_server_splitter/scripts/codesurfer/bs-
    script.stk /home/aspire/license/build/SC06.01/csurf-project//project -args /home/aspire/
    license/build/SC06.01/license.c.i -c 343 -v dd1 -b
16/10/25 16:37:49 - STEP7.d: moving fact files generated by codesurfer analysis to /home/aspire/
    license/build/SC06.01/facts/ folder
16/10/25 16:37:49 - STEP7.d: mv *.facts /home/aspire/license/build/SC06.01/facts/
16/10/25 16:37:49 - STEP8: converting pragmas to ASPIRE format (again)
16/10/25 16:37:49 - STEP8: /opt/client_server_splitter/scripts/pragma_conversion.py /home/aspire/
    license/build/SC06.01/license.c.i /home/aspire/license/build/SC06.01/license.c.i.4626.tmp.1
```

```
16/10/25 16:37:49 - STEP9: handling fact files (filtering and generation of new facts)
16/10/25 16:37:49 - STEP9: /opt/client_server_splitter/fact_handler.sh /home/aspire/license/build
    /SC06.01/license.c.i /home/aspire/license/build/SC06.01/facts/
16/10/25 16:37:49 - STEP9.a: cleaning facts
16/10/25 16:37:49 - STEP9.a: /opt/client_server_splitter/scripts/fact_cleaner.py /home/aspire/
    license/build/SC06.01/facts//license.c.i.uses /home/aspire/license/build/SC06.01/facts//
    license.c.i.uses.tmp
16/10/25 16:37:49 - STEP9.a: mv /home/aspire/license/build/SC06.01/facts//license.c.i.uses.tmp /
    home/aspire/license/build/SC06.01/facts//license.c.i.uses
16/10/25 16:37:49 - STEP9.a: /opt/client_server_splitter/scripts/fact_cleaner.py /home/aspire/
    license/build/SC06.01/facts//license.c.i.defuses /home/aspire/license/build/SC06.01/facts//
    license.c.i.defuses.tmp
16/10/25 16:37:49 - STEP9.a: mv /home/aspire/license/build/SC06.01/facts//license.c.i.defuses.tmp
     /home/aspire/license/build/SC06.01/facts//license.c.i.defuses
16/10/25 16:37:49 - STEP9.a: /opt/client_server_splitter/scripts/fact_cleaner.py /home/aspire/
    license/build/SC06.01/facts//license.c.i.defs /home/aspire/license/build/SC06.01/facts//
    license.c.i.defs.tmp
16/10/25 16:37:49 - STEP9.a: mv /home/aspire/license/build/SC06.01/facts//license.c.i.defs.tmp /
    home/aspire/license/build/SC06.01/facts//license.c.i.defs
16/10/25 16:37:49 - STEP9.b: applying fact merging
16/10/25 16:37:49 - STEP9.b: /opt/client_server_splitter/scripts/txl/fact-generator.x -o /home/
    aspire/license/build/SC06.01/facts//license.c.i.fullslice /home/aspire/license/build/SC06.01/
    facts//license.c.i.fullslice - -uses /home/aspire/license/build/SC06.01/facts//license.c.i.
    declarations -defs /home/aspire/license/build/SC06.01/facts//license.c.i.pdefs -pointers /
    home/aspire/license/build/SC06.01/facts//license.c.i.pointers -filename "/home/aspire/license
    /build/SC06.01/license.c.i"
16/10/25 16:37:49 - STEP9.c: generating fact files for client slicing
16/10/25 16:37:49 - STEP9.c: grep license.c.i /home/aspire/license/build/SC06.01/facts//license.c
    .i.fullslice > /home/aspire/license/build/SC06.01/facts//license.c.i.slice
16/10/25 16:37:49 - STEP9.c: sed 's/.*| //g' /home/aspire/license/build/SC06.01/facts//license.c.
    i.fullslice | sort | uniq > /home/aspire/license/build/SC06.01/facts//license.c.i.extern
16/10/25 16:37:49 - STEP9.d: extracting extra variables to be sent
16/10/25 16:37:49 - STEP9.d: /opt/client_server_splitter/scripts/txl/variable-extractor.x -s 1000
     -o /home/aspire/license/build/SC06.01/license.c.i /home/aspire/license/build/SC06.01/license
    .c.i - -slice /home/aspire/license/build/SC06.01/facts//license.c.i.slice -crit /home/aspire/
    license/build/SC06.01/facts//license.c.i.criterion -barrlines /home/aspire/license/build/SC06
    .01/facts//license.c.i.barriers -barriers /home/aspire/license/build/SC06.01/facts//license.c
    .i.barrier_vars -vars /home/aspire/license/build/SC06.01/facts//license.c.i.vars -defs /home/
    aspire/license/build/SC06.01/facts//license.c.i.defs -uses /home/aspire/license/build/SC06
    .01/facts//license.c.i.uses -defuses /home/aspire/license/build/SC06.01/facts//license.c.i.
    defuses -calls /home/aspire/license/build/SC06.01/facts//license.c.i.calls -types /home/
    aspire/license/build/SC06.01/facts//license.c.i.types -homedir /home/aspire/license/build/
    SC06.01/facts// -filename license.c.i
16/10/25 16:37:49 - STEP10: generating the client-side code
16/10/25 16:37:49 - STEP10: /opt/client_server_splitter/client-generator.sh -l /home/aspire/
    license/build/SC07/log/license.c.i.json /home/aspire/license/build/SC06.01/license.c.i.4626.
    tmp.1 /home/aspire/license/build/SC06.01/facts/ /home/aspire/license/build/SC07/
16/10/25 16:37:49 - STEP10.a: extracting client-side code
16/10/25 16:37:49 - STEP10.a: /opt/client_server_splitter/scripts/txl/extract-client-intra.x -s
    1000 -o /home/aspire/license/build/SC06.01/license.c.client /home/aspire/license/build/SC06
    .01/license.c.i.4626.tmp.1 - -slice /home/aspire/license/build/SC06.01/facts//license.c.i.
    slice -crit /home/aspire/license/build/SC06.01/facts//license.c.i.criterion -barrlines /home/
    aspire/license/build/SC06.01/facts//license.c.i.barriers -barriers /home/aspire/license/build
    /SC06.01/facts//license.c.i.barrier_vars -vars /home/aspire/license/build/SC06.01/facts//
    license.c.i.vars -defs /home/aspire/license/build/SC06.01/facts//license.c.i.defs -uses /home
    /aspire/license/build/SC06.01/facts//license.c.i.uses -defuses /home/aspire/license/build/
    SC06.01/facts//license.c.i.defuses -calls /home/aspire/license/build/SC06.01/facts//license.c
    .i.calls -types /home/aspire/license/build/SC06.01/facts//license.c.i.types -extern_vars /
    home/aspire/license/build/SC06.01/facts//license.c.i.other_variables -homedir /home/aspire/
    license/build/SC06.01/facts// -filename license.c.i -l /home/aspire/license/build/SC07/log/
    license.c.i.json
16/10/25 16:37:50 - STEP10.b: extracting client-side code from other files
16/10/25 16:37:50 - STEP10.b: found currently analysed file: "/home/aspire/license/build/SC06.01/
    license.c.i" no actions taken
16/10/25 16:37:50 - STEP10.c: applying inclusion resolution
16/10/25 16:37:50 - STEP10.c: /opt/client_server_splitter/scripts/txl/resolve-inclusions.x -s
    1000 -o /home/aspire/license/build/SC07//license.c.i /home/aspire/license/build/SC06.01/
    license.c.client - -library /opt/client_server_splitter/libraries/client/client_declarations.
    h
16/10/25 16:37:50 - STEP10.d: converting ASPIRE pragmas to original format (again)
16/10/25 16:37:50 - STEP10.d: /opt/client_server_splitter/scripts/pragma_conversion.py -r /home/
    aspire/license/build/SC07//license.c.i /home/aspire/license/build/SC07//license.c.i.tmp.1
16/10/25 16:37:50 - STEP11: generating the server-side code
16/10/25 16:37:50 - STEP11: /opt/client_server_splitter/server-generation.sh /home/aspire/license
    /build/SC06.01/license.c.i.4626.tmp.1 /home/aspire/license/build/SC06.01/facts/ /home/aspire/
```

```
    license/build/SCS01/
16/10/25 16:37:50 - STEP11.a: extracting server-side code
16/10/25 16:37:50 - STEP11.a: /opt/client_server_splitter/scripts/txl/extract-server-ASCL.x -s
    1000 -o /home/aspire/license/build/SC06.01/license.c.i.server /home/aspire/license/build/SC06
    .01/license.c.i.4626.tmp.1 - -slice /home/aspire/license/build/SC06.01/facts//license.c.i.
    slice -crit /home/aspire/license/build/SC06.01/facts//license.c.i.criterion -barriers /home/
    aspire/license/build/SC06.01/facts//license.c.i.barrier_vars -vars /home/aspire/license/build
    /SC06.01/facts//license.c.i.vars -map /home/aspire/license/build/SC06.01/facts//license.c.i.
    replacement_map -defs /home/aspire/license/build/SC06.01/facts//license.c.i.defs -uses /home/
    aspire/license/build/SC06.01/facts//license.c.i.survived_uses -fun /home/aspire/license/build
    /SC06.01/facts//license.c.i.fun_decl -ruses /home/aspire/license/build/SC06.01/facts//license
    .c.i.uses -extern_vars /home/aspire/license/build/SC06.01/facts//license.c.i.other_variables
    -calls /home/aspire/license/build/SC06.01/facts//license.c.i.calls -types /home/aspire/
    license/build/SC06.01/facts//license.c.i.types
int dd2;
time_t t;
struct tm tm;
16/10/25 16:37:50 - STEP11.b: applying server injection from other sources
16/10/25 16:37:50 - STEP11.b: found currently analysed file: "/home/aspire/license/build/SC06.01/
    license.c.i" no actions taken
16/10/25 16:37:50 - STEP11.c: applying slice injection to server
16/10/25 16:37:50 - STEP11.c: /opt/client_server_splitter/scripts/txl/inject-slice-to-server-ASCL
    .x -s 1000 -o /home/aspire/license/build/SCS01//slice.c /home/aspire/license/build/SC06.01/
    license.c.i.server - -barriers /home/aspire/license/build/SC06.01/facts//license.c.i.
    barrier_vars -vars /home/aspire/license/build/SC06.01/facts//license.c.i.vars -map /home/
    aspire/license/build/SC06.01/facts//license.c.i.replacement_map -uses /home/aspire/license/
    build/SC06.01/facts//license.c.i.survived_uses -extern_vars /home/aspire/license/build/SC06
    .01/facts//license.c.i.other_variables -filename license.c.i
16/10/25 16:37:50 - STEP11.d: converting ASPIRE pragmas to original format
16/10/25 16:37:50 - STEP11.d: /opt/client_server_splitter/scripts/pragma_conversion.py -r /home/
    aspire/license/build/SCS01//slice.c /home/aspire/license/build/SCS01//slice.c.tmp2
. SLP06_03_TRANSFORMATION
  transform      /home/aspire/license/build/SC06.01/library.c.i
  into           /home/aspire/license/build/SC07/library.c.i
16/10/25 16:37:50 - STEP7: no configuration file found for /home/aspire/license/build/SC06.01/
    library.c.i
16/10/25 16:37:50 - STEP7: copying /home/aspire/license/build/SC06.01/library.c.i to /home/aspire
    /license/build/SC07/
```

SLP08: SC07 → codeguard transformations → SC08

```
  create         /home/aspire/license/build/SC08
. SLP08_01_CG
  codeguard      /home/aspire/license/build/SC07/accl-message-wrapper.c.i
  into           /home/aspire/license/build/SC08/accl-message-wrapper.c.i
. SLP08_01_CG
  codeguard      /home/aspire/license/build/SC07/license.c.i
  into           /home/aspire/license/build/SC08/license.c.i
. SLP08_01_CG
  codeguard      /home/aspire/license/build/SC07/library.c.i
  into           /home/aspire/license/build/SC08/library.c.i
```

SLP09: SC08 → anti-cloning transformations → SC09 (traversed)
Anti-cloning is configured to be traversed in the ACTC configuration file for this application. No transformations are applied in this step, source files are copied directly to the destination folder.

```
  create         /home/aspire/license/build/SC09
. SLP09_AC
  copy           /home/aspire/license/build/SC08/accl-message-wrapper.c.i
  into           /home/aspire/license/build/SC09/accl-message-wrapper.c.i
. SLP09_AC
  copy           /home/aspire/license/build/SC08/license.c.i
  into           /home/aspire/license/build/SC09/license.c.i
. SLP09_AC
  copy           /home/aspire/license/build/SC08/library.c.i
  into           /home/aspire/license/build/SC09/library.c.i
. SLP10
  create         /home/aspire/license/build/SC10
. SLP10
  copy           /home/aspire/license/build/SC09/accl-message-wrapper.c.i
  into           /home/aspire/license/build/SC10/accl-message-wrapper.c.i
. SLP10
  copy           /home/aspire/license/build/SC09/license.c.i
  into           /home/aspire/license/build/SC10/license.c.i
. SLP10
  copy           /home/aspire/license/build/SC09/library.c.i
```

```
  into          /home/aspire/license/build/SC10/library.c.i
. SLP11
  create        /home/aspire/license/build/SC11
. SLP11
  copy          /home/aspire/license/build/SC10/accl-message-wrapper.c.i
  into          /home/aspire/license/build/SC11/accl-message-wrapper.c.i
. SLP11
  copy          /home/aspire/license/build/SC10/license.c.i
  into          /home/aspire/license/build/SC11/license.c.i
. SLP11
  copy          /home/aspire/license/build/SC10/library.c.i
  into          /home/aspire/license/build/SC11/library.c.i
. SLP12
  copy          /home/aspire/license/build/SC11/accl-message-wrapper.c.i
  into          /home/aspire/license/build/SC12/accl-message-wrapper.c.i
. SLP12
  copy          /home/aspire/license/build/SC11/license.c.i
  into          /home/aspire/license/build/SC12/license.c.i
. SLP12
  copy          /home/aspire/license/build/SC11/library.c.i
  into          /home/aspire/license/build/SC12/library.c.i
```

SLP04: SC09 → annotation extraction → D01
Annotations are extracted from the source code files and stored in a separate annotations.json file in the D01 directory. These annotations are later used by the binary to binary transformations.

```
  create        /home/aspire/license/build/D01
. SLP04_EXTRACT
  extract annot /home/aspire/license/build/SC12/accl-message-wrapper.c.i
  into          /home/aspire/license/build/D01/accl-message-wrapper.c.i.json
. SLP04_EXTRACT
  extract annot /home/aspire/license/build/SC12/license.c.i
  into          /home/aspire/license/build/D01/license.c.i.json
/opt/annotation_extractor/convert_pragmas.py /home/aspire/license/build/SC12/license.c.i /tmp
    /4896.i
. SLP04_EXTRACT
  extract annot /home/aspire/license/build/SC12/library.c.i
  into          /home/aspire/license/build/D01/library.c.i.json
. SLP04_MERGE
  merge         /home/aspire/license/build/D01/license.c.i.json
                /home/aspire/license/build/D01/library.c.i.json
                /home/aspire/license/build/D01/accl-message-wrapper.c.i.json
  into          /home/aspire/license/build/D01/annotations.json
```

Start of source to binary transformations
SC09 → compiler → BC08

```
  create        /home/aspire/license/build/BC08
. COMPILE_C
  compile       /home/aspire/license/build/SC12/accl-message-wrapper.c.i
  into          /home/aspire/license/build/BC08/accl-message-wrapper.c.i.o
/home/aspire/license/build/SC12/accl-message-wrapper.c.i:2080:12: warning: declaration of built-
    in function '__sigsetjmp' requires inclusion of the header <setjmp.h> [-Wbuiltin-requires-
    header]
extern int __sigsetjmp (struct __jmp_buf_tag *__env, int __savemask) __attribute__ ((__nothrow__)
    );
        ^
/home/aspire/license/build/SC12/accl-message-wrapper.c.i:3851:79: warning: incompatible pointer
    types passing 'char (*)[1024]' to parameter of type 'char *' [-Wincompatible-pointer-types]
   if (0 == acclWebSocketExchange(context, outBufferLength, outBuffer, 1024, &response_buffer)) {
                                                                              ^~~~~~~~~~~~~~~~
/home/aspire/license/build/SC12/accl-message-wrapper.c.i:3674:9: note: passing argument to
    parameter 'response' here
  char* response
        ^
2 warnings generated.
. COMPILE_C
  compile       /home/aspire/license/build/SC12/license.c.i
  into          /home/aspire/license/build/BC08/license.c.i.o
. COMPILE_C
  compile       /home/aspire/license/build/SC12/library.c.i
  into          /home/aspire/license/build/BC08/library.c.i.o
```

compile ACCL libs → BC08/accl
compile the ACCL libraries using the server address provided in the ACTC configuration json file.

```
. COMPILE_ACCL
  create         /home/aspire/license/build/BC08/accl
. COMPILE_ACCL
  compile        /opt/ACCL/src/accl.c
  into           /home/aspire/license/build/BC08/accl/accl.c.o
/opt/ACCL/src/accl.c: In function 'callback_accl_communication':
/opt/ACCL/src/accl.c:790:21: warning: 'user_context' may be used uninitialized in this function
    [-Wmaybe-uninitialized]
   if (user_context->wait_for_response) {
                ^
```

BC08 → linker → BC02
Link the generated binary files together with the ACCL binary and other external libraries required for the protection techniques.

```
  create         /home/aspire/license/build/BC02
. LINK
  link           /home/aspire/license/build/BC08/accl-message-wrapper.c.i.o
                 /home/aspire/license/build/BC08/library.c.i.o
                 /home/aspire/license/build/BC08/license.c.i.o
                 /home/aspire/license/build/BC08/accl/accl.c.o
                 /opt/3rd_party/libwebsockets/linux/lib/libwebsockets.a
                 /opt/3rd_party/curl/linux/lib/libcurl.a
                 /opt/3rd_party/openssl/linux/lib/libssl.a
                 /opt/3rd_party/openssl/linux/lib/libcrypto.a
  into           /home/aspire/license/build/BC02/a.out
```

BLP03: BC03 + BC08 + D01 → diablo linker → BC04

```
  create         /home/aspire/license/build/BC04
. BLP03_LINK
  link           /home/aspire/license/build/D01/annotations.json
                 /home/aspire/license/build/BC08/accl-message-wrapper.c.i.o
                 /home/aspire/license/build/BC08/library.c.i.o
                 /home/aspire/license/build/BC08/license.c.i.o
                 /home/aspire/license/build/BC08/accl/accl.c.o
                 /opt/3rd_party/libwebsockets/linux/lib/libwebsockets.a
                 /opt/3rd_party/curl/linux/lib/libcurl.a
                 /opt/3rd_party/openssl/linux/lib/libssl.a
                 /opt/3rd_party/openssl/linux/lib/libcrypto.a
  into           /home/aspire/license/build/BC04/c.out
```

BLP04: BC04 + D01 (+ BC08 + BC03 + BLC02) → diablo obfuscator → BC05

```
  create         /home/aspire/license/build/BC05
. BLP04_OBFUSCATE
  obfuscate      /home/aspire/license/build/D01/annotations.json
                 /home/aspire/license/build/BC04/c.out
  into           /home/aspire/license/build/BC05/d.out
```

M01: Metrics collection
Collect metrics from each compilation phase configured in the ACTC configuration json file and aggregates them in the M01 directory.

```
  create         /home/aspire/license/build/M01/
. M01_COLLECT
  create         /home/aspire/license/build/M01/BC05/profiles
. M01_COLLECT
  create         /home/aspire/license/build/M01/BC02_SP/profiles
. M01_COLLECT
  create         /home/aspire/license/build/M01/BC05_DYN
. M01_COLLECT
  create         /home/aspire/license/build/M01/BC05
. M01_COLLECT
  copy           /home/aspire/license/build/BC05/d.out.stat_complexity_info
  into           /home/aspire/license/build/M01/BC05/d.out.stat_complexity_info
. M01_COLLECT
  copy           /home/aspire/license/build/BC05/d.out.stat_regions_complexity_info
  into           /home/aspire/license/build/M01/BC05/d.out.stat_regions_complexity_info
. M01_COLLECT
  create         /home/aspire/license/build/M01/BC02_DYN
. M01_COLLECT
  create         /home/aspire/license/build/M01/BC02_SP
```

12.7 Graphical Representation of the ACTC Compilation Process

A graphical representation of the ACTC compilation process can be generated by execution the ACTC with the -p flag. The output can be found in the build folder and is shown in Figures 29 and 30. Each transformation is represented by an arrow between its source files and the output the transformation produces.

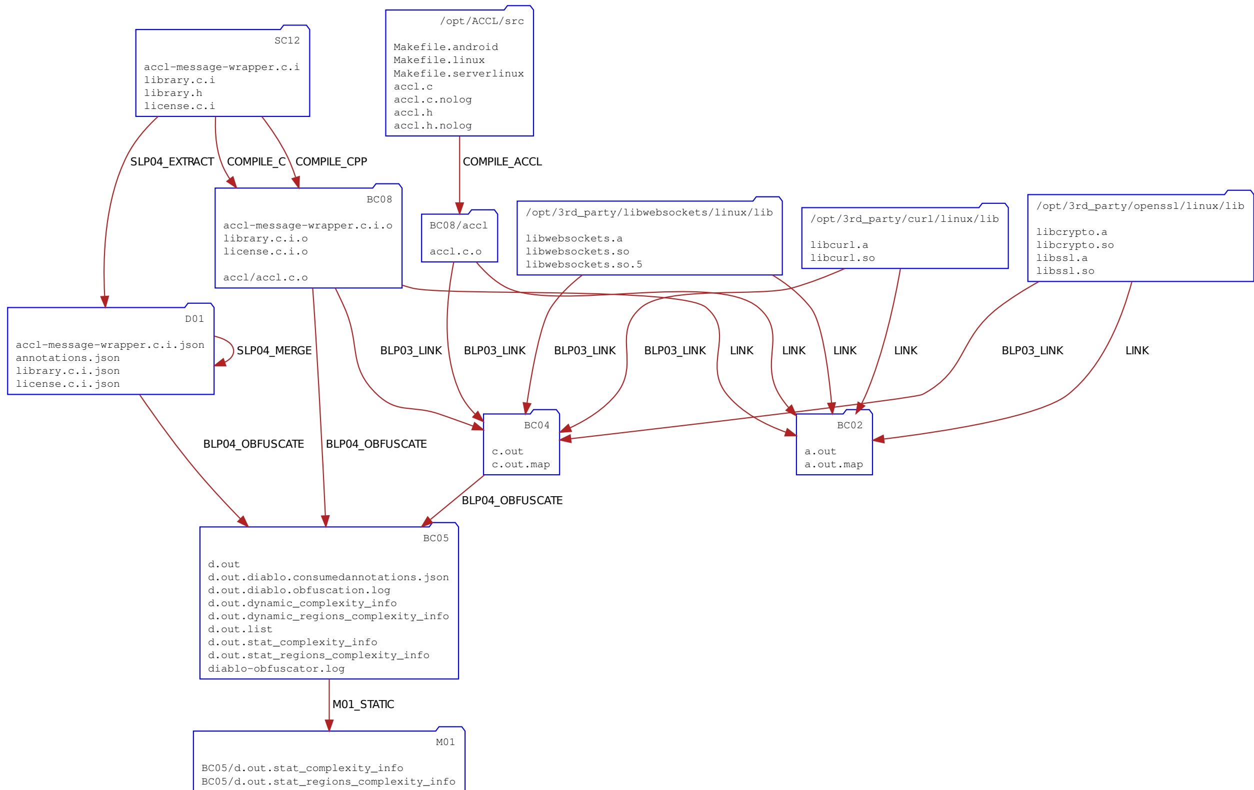

Figure 29: Graphical representation of the binary part of the ACTC compilation process.

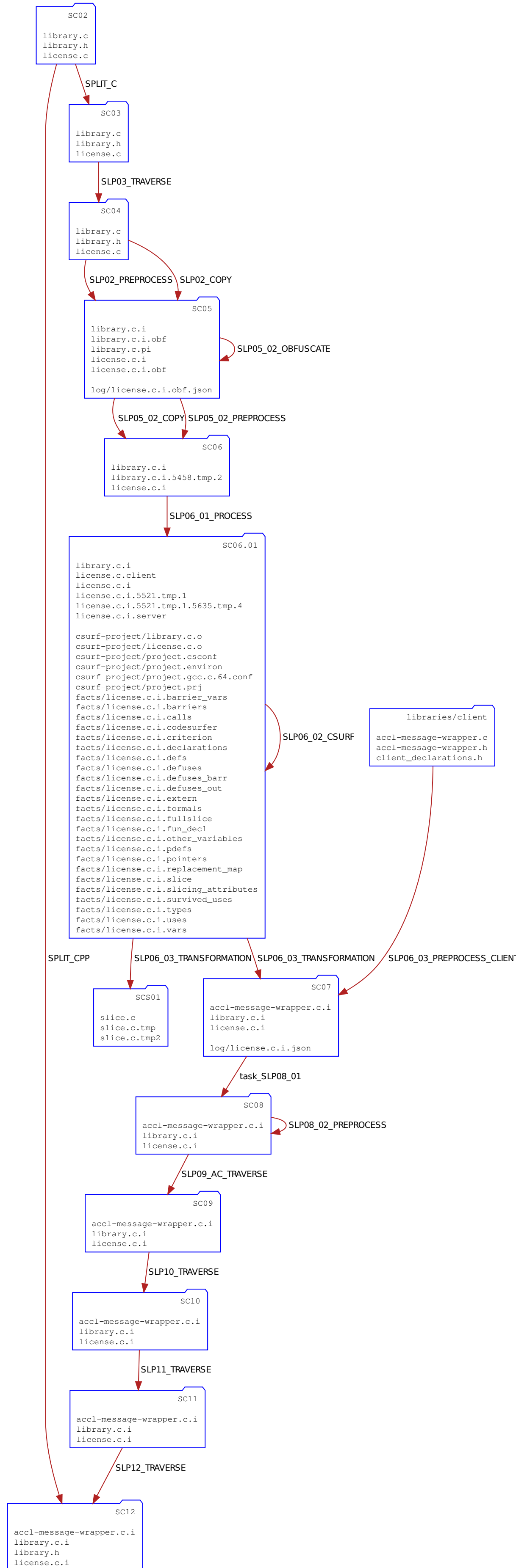

Figure 30: Graphical representation of the source part of the ACTC compilation process.

12.8 Result of Source Code Transformations

The resulting pre-processed source code after source code transformations have been applied is shown below. This code is produced by the last enabled source code transformation step, which in this case is the code guard transformation (SLP08). For clarity, the included files and external code are omitted and only the main function is shown.

The source to source protection techniques, data obfuscation and client-server code splitting have been applied, and the annotations have been removed from the source code as they are no longer required. The remaining annotations for binary code obfuscation, anti-debugging and code guards (all binary code transformations) are still present in the source code and have been extracted by the annotation extraction tool (SLP04) to be used by the binary to binary protection techniques.

```
int main (int argc, char **argv) {
send_arglen_to_server (argc);
send_args_to_server (argc, argv);
send_initial_message ();
#pragma ASPIRE begin protection (obfuscations,
  enable_obfuscation(branch_function:percent_apply=10,
  opaque_predicate:percent_apply=10))
int day1;
day1 = (0) ^ 35;
int mon1;
mon1 = (0) ^ 35;
int year1;
year1 = (0) ^ 35;
int day2;
int mon2;
int year2;
int ref;
int dd1;
int dd2;
int delta;
int i;
time_t t;
t = time (0);
send_value_to_server (500, & t, sizeof (t));
struct tm tm;
tm = * localtime (& t);
send_value_to_server (501, & tm, sizeof (tm));
#pragma ASPIRE begin protection(anti_debugging)
send_value_to_server (600, & argv, sizeof (argv));
for (i = 0; argv1i != '\0'; i++) {
send_value_to_server (601, & argv, sizeof (argv));
day1 = (10 * (day1 ^ 35) + (char) argv1i - '0') ^ 35;
}
send_value_to_server (602, & argv, sizeof (argv));
for (i = 0; argv2i != '\0'; i++) {
send_value_to_server (603, & argv, sizeof (argv));
mon1 = (10 * (mon1 ^ 35) + (char) argv2i - '0') ^ 35;
}
for (i = 0; i < 4; i++) {
send_value_to_server (604, & argv, sizeof (argv));
year1 = (10 * (year1 ^ 35) + (char) argv3i - '0') ^ 35;
}
#pragma ASPIRE end
;
#pragma ASPIRE begin protection(guarded_region,label(GR1))
day2 = (tm.tm_mday) ^ 35;
mon2 = (tm.tm_mon + 1) ^ 35;
year2 = (tm.tm_year + 1900) ^ 35;
ref = (year1 ^ 35) ^ 35;
dd1 = (0) ^ 35;
dd1 = (days_at_month (mon1 ^ 35)) ^ 35;
send_value_to_server (1, & dd1, sizeof (dd1));
for (i = (ref ^ 35); i < (year1 ^ 35); i++) {
if (i % 4 == 0) {
synch_with_server (1);
}
}
synch_with_server (2);
```

```
dd2 = (0) ^ 35;
for (i = (ref ^ 35); i < (year2 ^ 35); i++) {
if (i % 4 == 0) {
dd2 = ((dd2 ^ 35) + (1)) ^ 35;
}
}
#pragma ASPIRE end
;
dd2 = (days_at_month (mon2 ^ 35)+(dd2 ^ 35)+(day2 ^ 35)
   +(((year2 ^ 35) - (ref ^ 35)) * 365)) ^ 35;
send_value_to_server (502, & dd2, sizeof (dd2));
delta = ((dd2 ^ 35)
   - (ask_value_from_server (1, 1) ^ 35)) ^ 35;
if ((0 <= (delta ^ 35)) && ((delta ^ 35) <= 30)) {
printf ("Yes %d\n", (dd2 ^ 35)
   - (ask_value_from_server (1, 2) ^ 35));
send_value_to_server (1, & dd1, sizeof (dd1));
send_value_to_server (502, & dd2, sizeof (dd2));
}
else {
printf ("No %d\n", (dd2 ^ 35)
   - (ask_value_from_server (1, 3) ^ 35));
send_value_to_server (1, & dd1, sizeof (dd1));
send_value_to_server (502, & dd2, sizeof (dd2));
}
send_last_message ();
return 0;
#pragma ASPIRE end
}
```

13 The ASPIRE Shared Build Environment

Section authors:
Bart Coppens, Bjorn De Sutter, Jens Van den Broeck (UGent)

To ensure that partners can test their contributions to the ACTC and to the ASPIRE demonstration in the exact same, shared build environment, UGent provides such a build environment to all ASPIRE partners. This build environment consists of a Virtual Machine (VM) image and an ASPIRE repository from which the machine can receive updates. The partners will use this VM to generate and to deploy the ACTC on their software to be protected, be it toy examples, larger benchmarks or the project use cases. The VM therefore contains patched versions of a compiler tool chain, as described in Section 7, as well as all other components of the ACTC described in this document.

The patched compiler tool chains are installed on the VM using its regular Linux package management facilities. They are installed from a password-protected ASPIRE package repository maintained at UGent. The CodeSurfer and TXL software packages do not use the package management facilities, but are downloaded as tarballs from a password-protected UGent-ASPIRE server. Because of its fixed VM base image and controlled package repository, this build environment will enable all partners to create reproducible builds.

The VM image, which can be run with both VirtualBox and VMWare, is based on a minimalist Debian 7.4 installation. As distributed to ASPIRE partners, the image also contains a customization script that should be executed by the VM's user on the first boot of the VM. This customization performs the following steps:

- Setting of user preferences: users can choose between the KDE and Gnome graphical desktop environments, and select the correct keyboard lay-out.
- Ask for ASPIRE credentials to access the password-protected package repository.
- CodeSurfer license information: if the ASPIRE partner installing the VM has access to a CodeSurfer license, the installation can be customized to use this license.

This customization process then installs the following software packages:

- A graphical desktop environment, as chosen by the user.
- QEMU to be able to test statically linked ARM binaries in the VM, without requiring an ARM board.
- Several patched compiler tool chains:
  - Linux ARMv7, GCC 4.6.4 with binutils 2.23.2 and eglibc 2.17;
  - Linux ARMv7, GCC 4.8.1 with binutils 2.23.2 and eglibc 2.17;
  - Linux ARMv7 (hardfloat), GCC 4.8.1 with binutils 2.23.2 and eglibc 2.17;
  - Linux LLVM/clang 3.2, 3.3 and 3.4 (to be used in conjunction with a Linux GCC tool chain);
  - Android 4.3 Jelly Bean (API level 18), GCC 4.8 with binutils 2.23.2 and bionic libc;
  - Android 4.3 Jelly Bean (API level 18), LLVM 3.3 and 3.4 (to be used in conjunction with an Android GCC tool chain).
- TXL and CodeSurfer, installed from tarballs downloaded from an ASPIRE server.
- Eclipse as a development environment.

After this customization, the VM image can be used to compile software and to deploy the ACTC.

For the open sourcing of ASPIRE prototypes, we will prepare a Docker container (www.docker.com) instead of a VM image.

Part III

The ASPIRE Decision Support System

This part presents the ADSS, the tool that find the combinations of protection that best mitigate the risks against the annotated application assets. Several step are necessary to find the golden combinations, and each step required to address specific research issues, as presented in Section 14. Moreover, this part describes in Section 15 the ADSS tool, which has been released with the deliverable D5.10.

14 The ADSS work-flow and research issues towards the golden combinations

Section authors:
Daniele Canavese, Leonardo Regano, Cataldo Basile (POLITO)

The ultimate goal of the ADSS is to provide the user an all-inclusive framework to protect software in an automated fashion. In order to achieve this ambitious goal, the ADSS performs a series of analysis passes to understand the code structure and derives several deductions via a number of (both standard and custom built) inferential engines to find the most relevant and best protections to apply, named golden combinations.

A preliminary work-flow of the ADSS was presented in Deliverable D5.01, Section 13.1. This section discusses the main steps that are performed by the ADSS in order to find the golden combinations. For each step, this section documents the research issues addressed are the solutions used to address them. The up to date version is shown in Figure 31.

The new ADSS work-flow consists of six consecutive phases:

phase 1. *source code analysis* — a static analysis on the source code of the vanilla application is executed in order to find the assets, their properties and the code structure;

phase 2. *attack paths detection* — the attack paths on the vanilla application are inferred by using the data gathered in the previous phase;

phase 3. *protection detection* — the suitable protections for each asset are chosen in order to block the previously discovered attack paths;

phase 4. *first level protections discovery* (or L1P for short) — the optimal combination of protections (the golden combinations) for maximizing the application security are computed;

phase 5. *second level protections discovery* (or L2P for short) — additional protections are added to the L1P golden combinations in order to confuse the attacker about the assets' location and further delay attacks (this step is optional and can be skipped);

phase 6. *solution deployment* — the chosen protection combination is used to derive the annotations so that the ACTC will enforce them.

14.1 Source code analysis

The first phase undertaken by the ADSS is the *source code analysis*, whose job is to locate the application parts[3], the assets and understand their relationships in order to create a suitable and

[3]Application parts are all the entities in the application to protect that have been already marked as assets or may deserve to be protected, like variables, functions, and code regions.

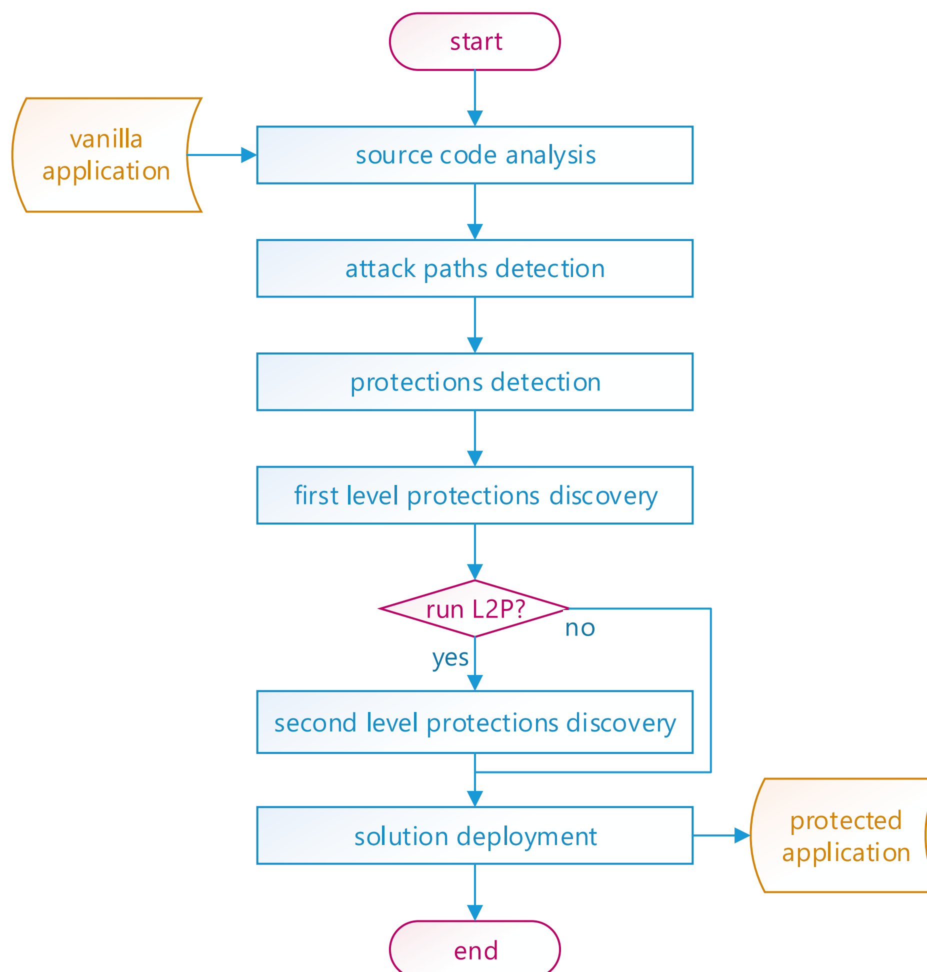

Figure 31: Work-flow of the ADSS.

complete AKB for all the remaining steps (see Deliverable D4.06, Sections 2 and 3, for more information about its content and organization). This phase performs various tasks at once, described in the following paragraphs.

14.1.1 Static code analysis

The ADSS' first task is to perform a static analysis of the application source code files and headers for extracting various information such as:

- the list of all the application parts (i.e., functions, their parameters and variables);
- the call graph of the functions;
- the data types, sizes and usages of all the variables and function parameters.

This is done primarily leveraging the CDT[4] (C Development Toolkit) framework, containing a very powerful C and C++ source files parser. In addition it uses some Perl scripts to parse the

[4]See `https://eclipse.org/cdt/`.

application's ACTC JSON file and extract some meaningful data (such as the recursive list of all the included headers, needed for correctly analyzing the files with the CDT parser).

Previous versions of the ADSS relied on the use of CodeSurfer[5], however, in the latest releases we opted to switch to the free and open-source CDT parser.

14.1.2 Annotation extraction

The next task extracts the annotated code regions, variables and the security properties that must be guaranteed on them (see Section 3 for more information about this subject). This task is performed by making again use of the CDT parser.

14.1.3 Execution of user-defined application-specific rules

The user can specify a set of custom rules to be executed in order to deduce some additional information. These rules can be freely inserted by the user in the UI via a custom DSL (see Section 4.2.5 of D5.13) and are used to provide additional inferential rules to the ADSS for deducing some relationships that are application-dependent and cannot be known a-priori. For instance, they can be used to specify that a custom function is used to encrypt data with AES-128 in CBC mode (this information can be used later in the ADSS work-flow for correctly placing the white-box cryptography technique annotations). More information and an example are available in Section 4.2.5 of D5.13.

The editor and the DSL parser are implemented with the Xtext[6] technology.

14.1.4 Vanilla application build

Finally, the ADSS triggers an initial ACTC build for extracting the metrics on the vanilla application. These data are used later during the L1P and L2P discovery phases (see Deliverable D4.06, Section 4, for more information about the software complexity metrics).

14.2 Attack paths detection

The second phase in the ADSS work-flow is to find all the attack paths against the assets. This stage is crucial since all the L1P protections will be placed in order to block the detected attack paths. Its sub-processes are described in the following sections.

14.2.1 Identification of the protection objectives

First, all the protection objective are computed. A *protection objective*, or *PO*, is a pair of an asset and one of its security requirement (e.g., $(function_1, integrity)$ or $(variable_1, confidentiality)$).

The POs are important since they are the end targets of the attacker (i.e., the security properties he wants to breach on the assets). In turn they became the targets of the protections (as the ADSS perspective is the protection).

14.2.2 Attack paths computation

During this phase, all the attack paths that can breach the POs are computed.

[5]See `https://www.grammatech.com/products/codesurfer`.

[6]See `http://www.eclipse.org/Xtext/`.

This task is largely implemented in SWI-Prolog[7] with some additional Java methods for managing the I/O between the ADSS and the Prolog engine. An initial description was sketched in Deliverable 5.01, Section 14 and Appendix F, and the same basic ideas were expanded in two ASPIRE publications [1, 9]. Nonetheless, a brief recap is available in the following paragraphs for the sake of completeness.

Each attack path is an ordered sequence of attack steps to be executed to breach one or more POs. The attack path discovery algorithm works with a backward chaining (or backward reasoning) approach. That is, the Prolog engine is asked to prove if and how an attacker can breach a PO (his goal) by knowing a set of initial facts (known to be true), i.e., the axioms. These axioms include assertions about the application structure and its assets such as 'the function $function_1$ accesses the variable $variable_1$' or 'the code region $region_1$ is contained in the function $function_1$'. By knowing these facts the Prolog engine can perform a search from the goals to the axioms in order to prove that a goal can be effectively reached (the PO can be breached) or not (the PO is safe). Figure 32 shows a simplified example of a proof graph for proving that the PO $(region_1, integrity)$ can be breached with a sequence of attack steps. In the graph the goal is the central red node labeled as $(region_1, integrity)$, the blue nodes with thick border are attack steps while the green rectangles are axioms. By enumerating all the possible paths to reach the goal PO the ADSS can infer all the attack paths, that, in the sample case presented before, are:

- statically locate $function_1$, statically locate $region_1$, statically change $region_1$;
- dynamically locate $function_1$, dynamically locate $region_1$, dynamically change $region_1$.

14.2.3 Attack steps classification

Finally, the ADSS classifies the attack steps of each attack path in one or more attack types (e.g., static tampering, dynamic data structure and code analysis). This classification is needed later to the ADSS in order to decide the proper protections for blocking an attack path.

Note that not all the attack steps are attacks, so they will lack a proper attack type. For instance, the steps 'setup server' or 'execute $function_1$' belong to this category of attack steps.

14.3 Protection detection

Once the attack paths are found, the ADSS next job is to infer the protections that can mitigate them. Note that from the ADSS point-of-view, an attack path is mitigated when at least one of its attack steps is mitigated. This simplifies the job in detecting how to mitigate all the attack steps.

This phase performs essentially only one pivotal task: finding the suitable applied protection instantiations. We recall that a *protection instantiation*, or *PI* for short, consists of a protection technique (e.g., anti-debugging) coupled with its configuration parameters. The same protection technique can offers multiple protection instantiations. For instance, the remote attestation technique can provide an instantiation where SHA1 is used and another one where RIPEMD is used. An *applied protection instantiation*, or *applied PI* for short, is a pair consisting of a protection instantiation and an application part (usually an asset). They represent the basic protection unit of the ADSS since they specify exactly where a protection should be applied and its implementation details.

Via a Java algorithm developed on purpose, all the feasible applied PIs are detected by taking into account all the related constraints, which concern:

- limitations on the application part type where techniques can be applied (e.g., white-box cryptography can only be applied on cryptographic keys);

[7] See `http://www.swi-prolog.org/`.

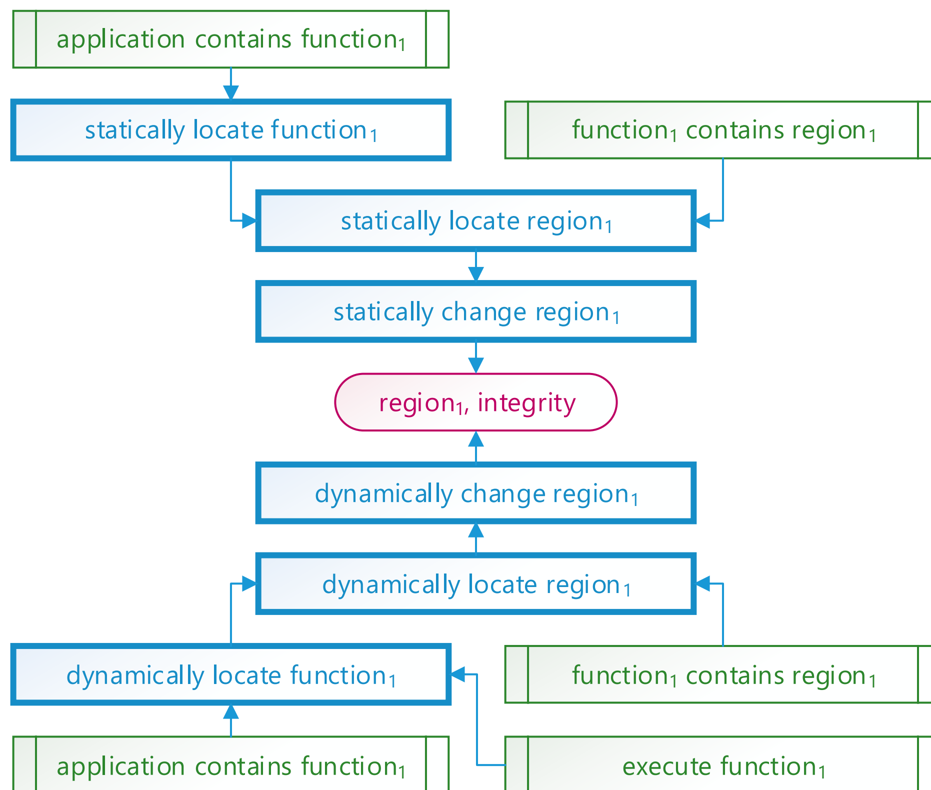

Figure 32: Example graph for proving $(region_1, integrity)$.

- effectiveness of the protection to preserve security requirements (e.g., binary obfuscation technique only protect the code confidentiality, not its integrity);
- protection-specific constraints (e.g., code mobility can only be applied to code regions spanning over an entire function);
- user-defined constraints on the maximum overheads (e.g., if the user sets overhead thresholds all the applied PIs that exceeding the threshold are not generated) and user preferences (e.g., discarding some techniques).

More information about this discovery procedure can also be found in an ASPIRE publication [9].

14.4 First level protections discovery

The only task of the protections detection phase is to find the applied PIs that may serve to protect the POs, but in isolation. The L1P discovery phase aims to find how to better combine them in order to maximize the mitigation of discovered attack paths against the application assets, also satisfying additional constraints derived from user constraints (such as the overhead limits). The final result is the list of the best possible protection combinations with the highest *ADSS protection indexes*, a score stating how safe a solution is (the ADSS protection Index). The protection combination with the highest protection index is the optimal one, that is the *golden combination*.

This phase is realized through two interconnected algorithms: the solution walker and the solution solver.

14.4.1 Solution walker

The *solution walker* computes all the feasible protection combinations, by taking into account all the user-defined and built-in constraints. An initial description is available in Deliverable D5.07, Section 5. The current algorithm is a slightly revised version, improved for maximizing the speed, with an one additional step w.r.t. the original one. Its flowchart is shown in Figure 33.

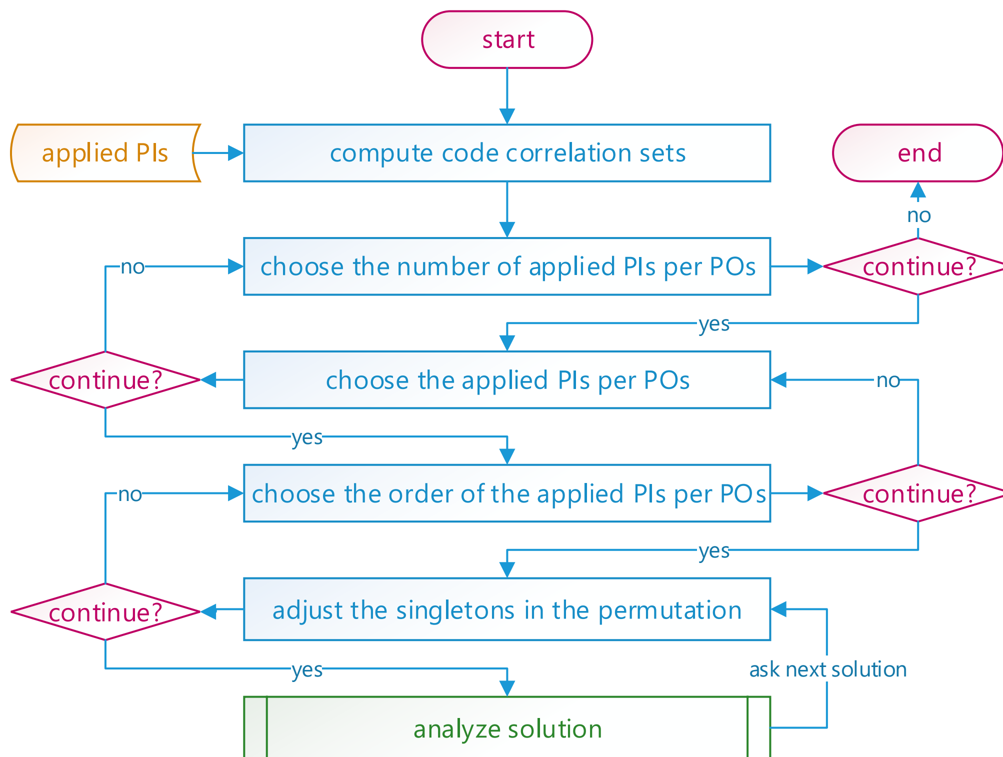

Figure 33: Simplified flow-chart of the solution walker algorithm.

In a nutshell, the algorithm has six steps:

step 1. *compute the code correlation sets* — code correlation sets are groups of application parts with some source code in common. They are used to speed-up the computation and guarantee coherent application of protections (more information is available in Deliverable D5.07, Section 5);

step 2. *choose the number of applied PIs per POs* — each time this step is called, it computes the next admissible tuple of integers specifying how many PIs must be put on a PO. By default each PO must be protected with at least one applied PI, in order to offer at least some degree of protection for the entire application. When all the possible Cartesian products of admissible integers are explored, there are no more solutions to find and the algorithm terminates, otherwise it hands over the selected tuples to the next step;

step 3. *choose the applied PIs per POs* — each time this step is called, it computes the next admissible unordered set of PIs for each PO. Note that the cardinalities of these sets are computed in the previous step. If all the possible applied PIs' sets are explored the algorithm goes back one step, otherwise it hands over to the next step the admissible unordered set of PIs for each PO;

step 4. *choose the order applied PIs per POs* — each time this step is called, the ADSS computes the next admissible order of the applied PIs for each PO (chosen in the previous step). If all the possible permutations are explored the algorithm goes back one step, otherwise it it hands over to the next step the admissible ordered set of PIs for each PO (i.e., a combination);

step 5. *adjust the singletons in the permutation* — each time this step is called, the ADSS computes the next admissible tuple of singleton protections[8] in the currently ordered list of applied PIs. For instance, if the previous step returns $\langle(PI_1, part_1), (PI_1, part_2), (PI_3, part_3)\rangle$ and PI_1 is related to a singleton protection with two PIs PI_1 and PI_2, this step will iterate over $\langle(PI_1, part_1), (PI_1, part_2), (PI_3, part_3)\rangle$, $\langle(PI_1, part_1), (PI_2, part_2), (PI_3, part_3)\rangle$, $\langle(PI_2, part_1), (PI_1, part_2), (PI_3, part_3)\rangle$ and $\langle(PI_2, part_1), (PI_2, part_2), (PI_3, part_3)\rangle$. If all the tuples of singletons are explored the algorithm goes back one step, otherwise it hands over the combination including singleton to the next step;

step 6. *evaluate solution* — the solution is evaluated with the solution solver and once its analysis is completed a new solution is asked by calling the previous step.

The step 2. and step 5. use a custom implementation of the loop-less reflected mixed-radix Gray generation algorithm [6], the step 3. uses the Chase's sequence algorithm [8] and the step 4. uses the lexicographic permutations with restricted prefixes algorithm [7].

14.4.2 Solution solver

The *solution solver* is an optimization solver that searches the solution space given by the solution walker in order to find the optimum, that is the golden combination. The current implementation is based on a heavily customized minimax tree [10], a game theoretical approach widely used in most of the state-of-the-art chess engines such as Stockfish[9] and the famous Deep Blue IBM super-computer [2]. To short describe the customizations, we depicted in Figure 34 the tree that the ADSS internally builds when looking for a solution. Note that the tree might be unbalanced due to the effects of pruning and reduction techniques.

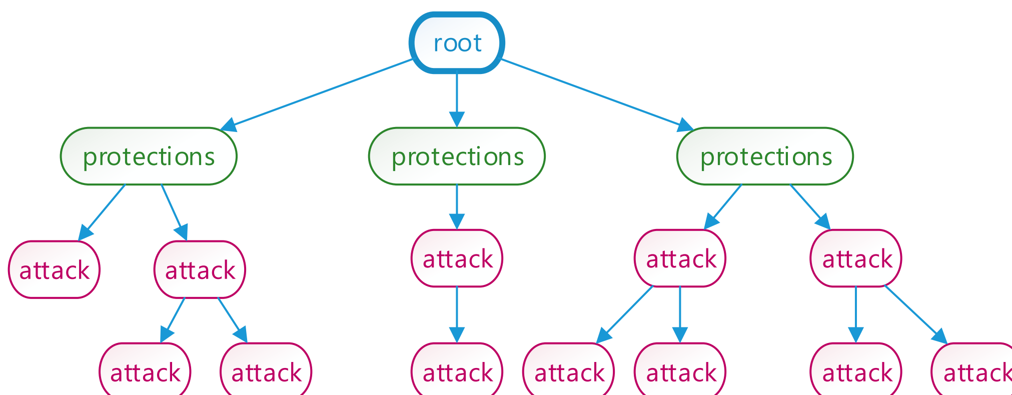

Figure 34: Minimax tree of the ADSS.

The root node represents the global maximum, i.e., the golden combination, the first level nodes represent all the suitable combinations (given by the solution walker) and all the remaining nodes

[8] We recall that a *singleton* protection is a protection that can be applied at most one time to an application part.
[9] See `https://stockfishchess.org/`.

are the attacks paths (replicated $h-1$ times if the tree height is h). This emulates a turn-based game where the first turn is for the defender (the first level nodes) and all the remaining turns are reserved for the attacker (all the other non-first level nodes). The ADSS performs a depth-first search on the tree and propagates the scores using the following rules:

- if the current node is a leaf, its security score (named the *ADSS protection index*) is computed by taking into account all the nodes in the path from the root to the leaf itself, that are a single protection combination and a list of attack paths. That means that each leaf is related to a protection combination attacked by a set of attack paths;
- if the current node is an internal node, but not the root, its score is the minimum of its children. Since the attacker goal is to minimize the security, he will likely choose the most successful attack path, that is the one that will lower the protection index the most;
- if the current node is the root, its score is the maximum of its children. This is due to the fact that the ADSS wants to maximize the security, that is it will choose the protection combination that will increase the protection index the most.

The result of the analysis is that the optimum (the tree root) is chosen as the most secure solution that mitigates the most dangerous list of attacks. Crucial to this computation is the leaf evaluation function that computes the protection index. This score is obtained by taking into account a number of different factors such as:

- the security of each protection in isolation (estimated by means of the software metrics);
- the security given by the synergy of protections (both for encouraged and discouraged interactions between the protections);
- the threat level and probability of the attack paths, which are influenced by the attacker expertise level, fully configurable in the UI;
- bonuses and penalties (for instance if the security of an asset is below a threshold a big penalty is triggered since the ADSS considers the asset as breached).

In order to speed-up the computation, the ADSS implements a more sophisticated version of the basic minimax search tree and several pruning and reduction techniques[10] that includes:

- alpha-beta pruning with aspiration windows;
- iterative deepening with transposition tables;
- razoring, futility margin, extended futility margin and reductions based on the node scores. These techniques are used to preventively stop the search if a node's score is too low or high. In these states, in fact, the application is highly likely/unlikely to be breached, that is the outcome of the attacks is clear, so that it does not need any further analysis.

In order to further speed-up the computations the search algorithm is still in active development. New pruning techniques and better search algorithms are investigated, and we are working (and we will continue after the end of the project) to add support for genetic algorithms to further reduce the search time.

[10] See `https://chessprogramming.wikispaces.com/`.

14.5 Second level protections discovery

The L2P discovery phase is used to add additional applied PIs to the protection combinations produced by the L1P algorithms. It is optional and can be skipped by the user, if the user chooses to.

The ultimate goal of this additional protection layer is not to directly increase the assets' security by protecting them with additional techniques (this is the L1P's job). The L2P phase indirectly increase the assets' security by rendering to attackers more difficult to identify the asset's locations. Some protection techniques are easy to spot since they leave a distinctive pattern recognizable in the protected code. For instance, the XOR masking introduces a lot of XOR operations in the code, somewhat signaling the attacker that a variable involved in a lot of such operations might be an asset, and hence containing valuable data. The L2P phase adds some additional decoy protections to the code that should increase the attacker's time for reverse engineering the application and its assets.

The new L2P applied PIs are generated by solving a custom MILP (Mixed Integer Linear Problem). The L2P phase supports by default the IBM CPLEX optimizer[11], but the user can also use another software by implementing a simple interface (see Section 15.2.4).

The ADSS actually makes use of three different techniques to increase the confusion level in the code.

The first one is the *replication* technique, where a new applied PI is created with a PI already existing in the L1P combination but on a code region created in a randomly chosen function. Figure 35 shows an example of replication where to the L1P applied PIs ($(PI_1, part_1)$, $(PI_2, part_1)$, $(PI_1, part_2)$ and $(PI_3, part_3)$) are added two L2P applied PIs ($(PI_2, part_4)$ and $(PI_3, part_5)$). Note that the L2P applied PIs are placed on two new application parts ($part_4$ and $part_5$) not in the list of the L1P application parts, but they make use of two PIs in the L1P PIs list.

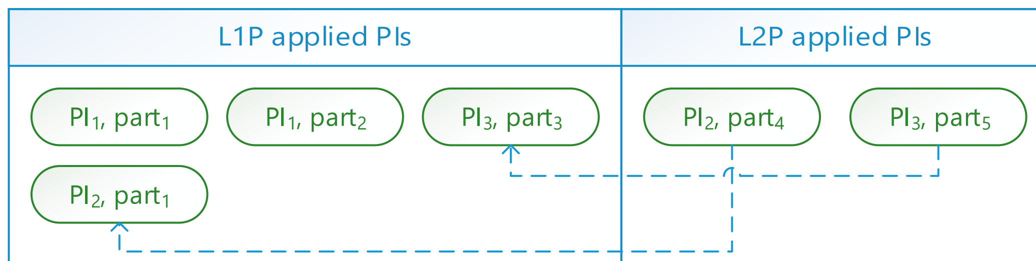

Figure 35: Example of the L2P replication technique.

The second one is the *enlargement* technique, where an already existing applied PI related to a code region is expanded, that is its covered area is made bigger. Its visual representation is shown in Figure 36. Obviously, the functions' boundaries are the maximum extension of a code region.

The third and final one is the *call graph extension*. Starting from a code region within a function, the area to protect is extended (by keeping the protections already decided for the asset) first to cover the whole function (as in the previous case), then it is virtually extended outside the function boundaries but not linearly in the binaries rather jumping to the point where the function is called. Practically, a new applied PI is created with the PI in the L1P combination and applied to a code region that includes the callers/callees code of the function that originally included the asset. Its idea is sketched in Figure 37. Note that this technique is only meaningful with specific protections.

[11]See https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

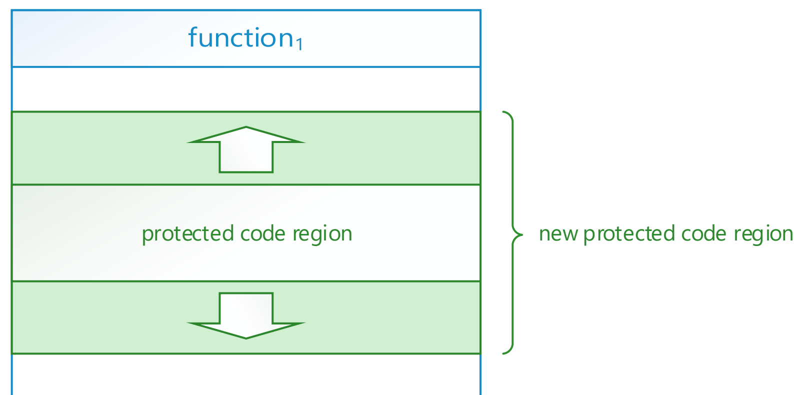

Figure 36: Example of the L2P enlargement technique.

For instance, it is useless for the binary obfuscation since the function calls are not masked and still visible. However, it is useful for the SoftVM technique since all the function invocations are removed from the application and moved to the virtual machine.

To increase the analysis speed of the L2P combinations we are currently actively working on a new algorithm based on genetic algorithms that will provide much faster computation times, albeit slightly sacrificing the search accuracy.

We also report here another aspect of the L2P that has been designed but is not yet available in the current version of the ADSS: cross protection, that is, the deployment of protections that protect the other deployed protections. Indeed, protections may strengthen each other not only when they are deployed in combination to protect specific security properties of an asset, but also when they are deployed with the explicit purpose of protecting other L1P protections deployed to actually protect assets. As an example, WBC may take advantage of integrity protection techniques, like remote attestation or code guards, in order to detect and counter the modifications that may be needed to attack WBC. Or anti-debugging can benefit from remote attestation to ensure that the anti-debugging is not disabled.

Cross protection introduces new assets, which are the pieces of code affected by the application of a protection (either at source or binary level, either modification of the original vanilla code or new inserted code). The addition of a new asset alters the order of the steps proposed in this workflow, as new assets should be considered very late, when L1P are already decided. Therefore, we decided to make cross protection part of the L2P phase, that is, we use for this operation only the resources that have not been saturated by L1P.

Since the metrics framework is executed on the vanilla application, we are currently not able to estimate the impact of protections when deployed on top of other protections. Nor it is possible to compute the metrics for a set of combinations big enough to be significative for the search space the ADSS has to explore. To cope with this issue, we are currently developing models that are able to foresee the impact of protections when applied to protect other protections based on a priori information about the transformations (insertions, modifications) the protections will perform on the vanilla code.

However, there is one thing that can be done with the current ADSS and ACTC: the protection owners can add explicit protection specific annotations to indicate how the added code must be protected. That is, there is no decision support to help users to decide how to protect the deployed protections, but the protection owner can make explicit decisions himself. The ADSS only needs

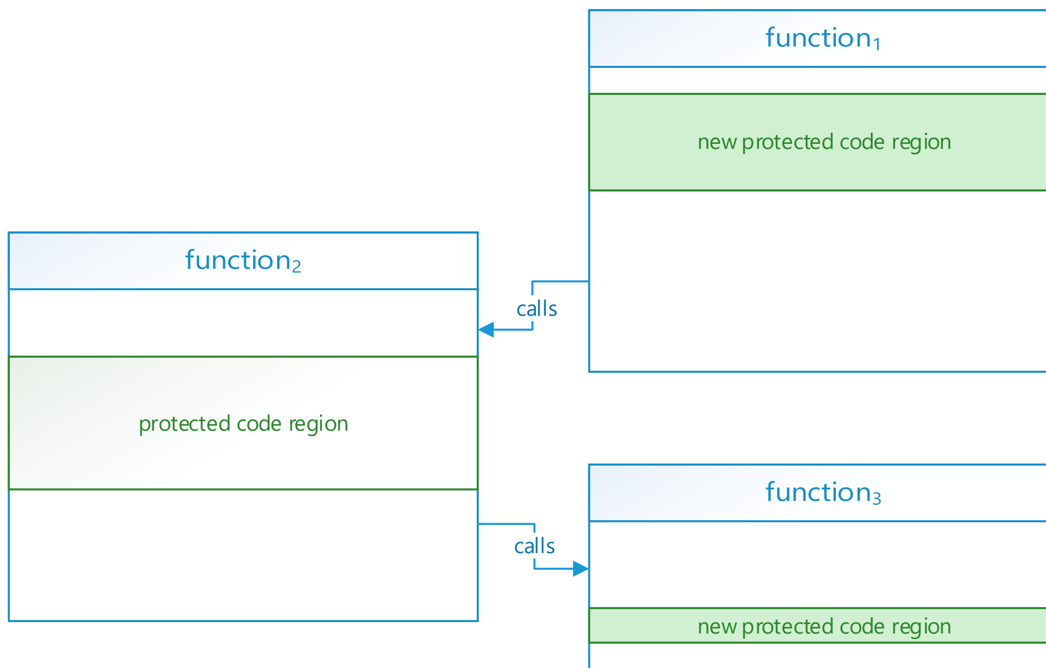

Figure 37: Example of the L2P call graph extension technique.

to consider the overheads added by the additional protections when estimating the impact of the application of a protection. Additional constraints may be added in case of conflicts among deployed L1P protections and explicitly selected additional protections.

14.6 Solution deployment

The last phase in the ADSS work-flow is to deploy the chosen combination of protections, that is creating the files needed for the ACTC to actually produce the protected binary. Note that the solution can be with or without L2P protections since only the L1P are mandatory.

This step is relatively straightforward and consists of:

1. generating a patch file and a JSON file for the ACTC (see Section 6.2);
2. triggering an ACTC build in order to generate the protected application.

15 The ADSS tool

Section authors:
Daniele Canavese, Leonardo Regano, Cataldo Basile (POLITO)

The following sections document the architecture of ADSS, the tool released with deliverable D5.10. Moreover they provide instructions to extend the ADSS functionalities. For installation information and general information on how to use the ADSS we refer to the deliverable D5.13. The deliverable D5.13 also contains information about the repository from which the ADSS can be downloaded.

15.1 The ADSS Architecture

The ADSS is a complex framework consisting of multiple building blocks distributed in several plug-ins. Table 4 lists some metrics and statistics of the ADSS.

Plug-ins	12
Additional JAR libraries used	22
Packages	58
Number of classes	338
Lines of code	47189

Table 4: Metrics of the ADSS.

15.1.1 Plug-ins

The ADSS consists of twelve plug-ins, whose dependencies are shown in Figure 38.

Six plug-ins store the core of the ADSS and are:

- `eu.aspire_fp7.adss` – all the classes for analyzing the source code, detecting the attacks and protecting the application reside here;
- `eu.aspire_fp7.adss.util` – a simple collection of general purpose utility methods used by several other plug-ins;
- `eu.aspire_fp7.adss.akb` – it contains all the classes related to the AKB;
- `eu.aspire_fp7.adss.rules` – it comprises the user-defined rules DSL Xtext grammar and its connection logic with the AKB;
- `eu.aspire_fp7.adss.rules.ide` – the parser for the user-defined rules language files;
- `it.polito.security.ontologies` – contains the code for managing ontologies, as the AKB is an ontology.

The UI is split in four additional plug-ins:

- `eu.aspire_fp7.adss.akb.ui` – it contains all the AKB editor pages in common between the ADSS Full and Light;
- `eu.aspire_fp7.adss.full.akb.ui` – all the ADSS Full specific editor pages are stored in here;

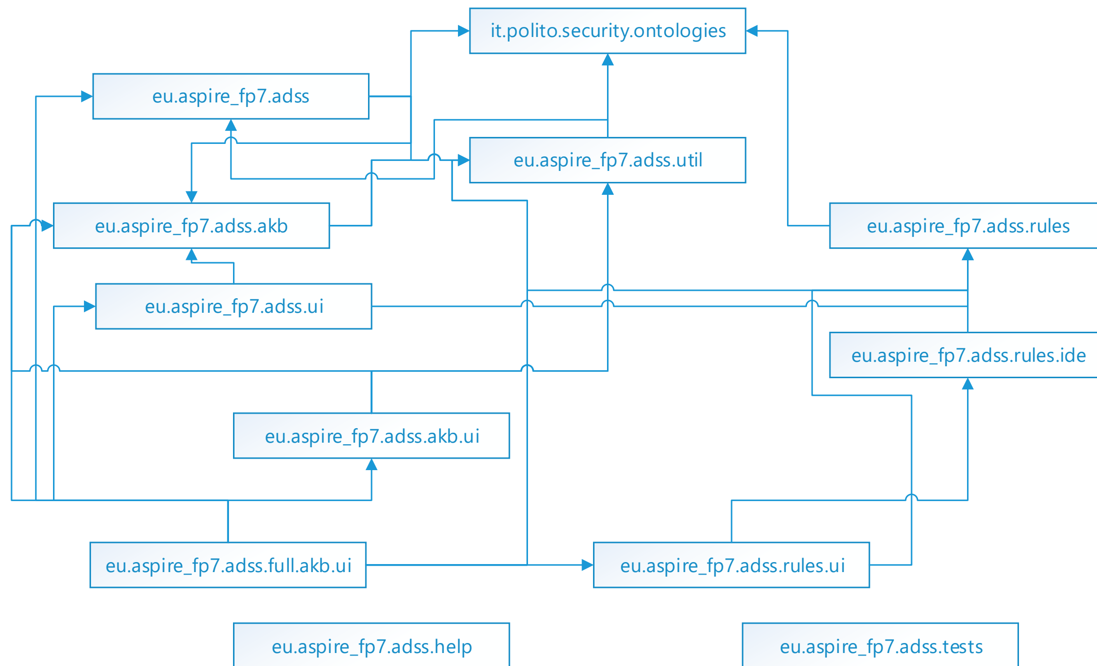

Figure 38: ADSS plug-in dependencies.

- `eu.aspire_fp7.adss.rules.ui` – the user-define rules editor interface;
- `eu.aspire_fp7.adss.ui` – a collection of utility methods related to the UI.

Finally, there are also two plug-ins not containing code:

- `eu.aspire_fp7.adss.help` – it contains the internal documentation of the ADSS and its manuals;
- `eu.aspire_fp7.adss.tests` – a collection of projects, used for testing the ADSS correctness.

15.1.2 Main components

Figure 39 sketches the interaction between the core components of the ADSS. Note that, for simplicity, this figure does not include the UI blocks.

Three components are commonly used in all the ADSS: the *ACTC connector*, which communicates with the ACTC for triggering the project builds, the *metrics framework*, used to gather the metrics computed by the ACTC or estimate them without an explicit build, and the *AKB utilities*, a collection of methods used to manage the AKB (e.g., to save, load and initialize the AKB from scratch).

The remaining components are used to build an enrichment framework for increasing the knowledge stored in the AKB.

The source code analysis phase is an enrichment module consisting of a *CDT connector*, whose task is to handle the communications with the CDT parser, and an *analyzer* that interprets the source code data gathered via CDT and saves the results into the AKB.

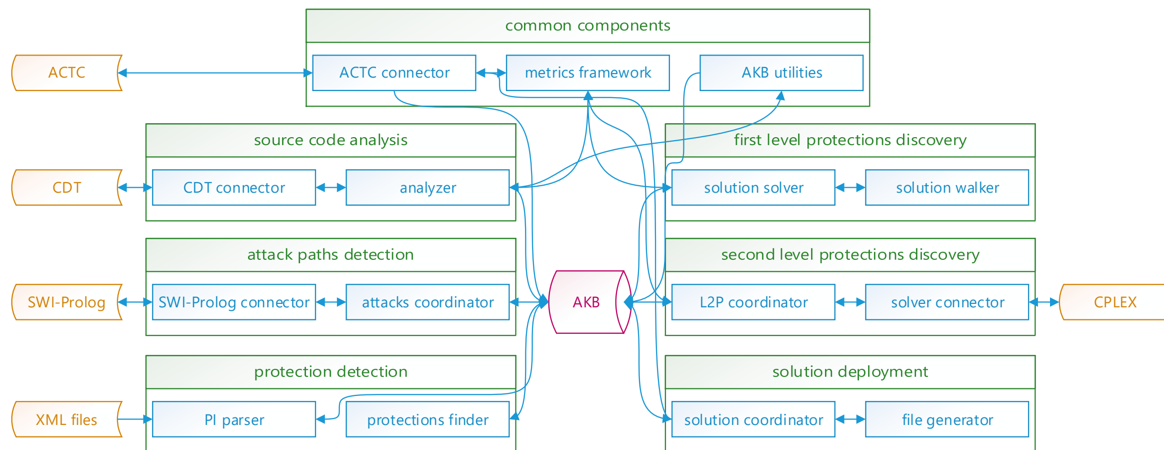

Figure 39: Architecture of the ADSS.

The attack paths detection phase uses a *SWI-Prolog connector* for executing the external Prolog engine and a relatively simple *attacks coordinator* that reads the application structure data from the AKB and enriches in it the found attack paths.

The protection detection phase comprehends two separate enrichment modules. The first one is the *PI parser*, a simple XML parser that reads a set of XML files storing the PI data (see Section 15.2.1) and stores all the information in the AKB. The second one is the *protection finder* used to create and save all the suitable applied PIs.

The first level protections discovery phase has two main components: a *solution walker*, used to produce and iterate over all the suitable combination of protections, and the *solution solver*, that picks the best combinations thanks to a minimax tree approach. More information is available in Section 14.4.

The second level protections discovery phase implements its functionalities via a *solver connector*, handling the communication with an external MILP solver, and an *L2P coordinator* that builds the linear model and interprets the results obtained via the external solver. Currently only the IBM CPLEX solver is supported, however, Section 15.2.4 details how to add the support for new solvers.

Finally, the solution deployment phase has a *file generator*, used to create the ADSS patch and JSON files (see Section 6.2), and a *solution coordinator* whose job is to actually deploy these files and triggers an ACTC build. Note that these components are not enrichment modules since they only read data from the AKB.

15.2 Expanding the AKB and the ADSS

In this section we provide the main information for extending the ADSS. In order to perform the extensions proposed here, a developer installation needs to be done, as presented in D5.13.

15.2.1 Adding new protection instantiations

The *protection instantiations* (see Section 32) are stored in the `protectionInstantiations` folder of the `eu.aspire-fp7.adss.akb` package, in form of XML files; there is a file for every protection, containing all its PIs. All XML files are validated against the XML schema contained in the file `protectionInstantiation.xsd`, located in the folder `schema` of the aforementioned package.

To add new PIs to the ADSS, relative to ADSS supported protections, the developer must:

1. create a new XML file;
2. inside the XML file, create the main element `protectionInstantiationsList`, adding the following attributes:
   - `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
   - `xmlns="http://protectionInstantiation.akb.adss.aspire_fp7.eu"`
   - `xsi:schemaLocation` set to the path of the PI XSD file;
3. inside the PI list element, create a `protectionInstantiation` element for every new PI, containing the following element:
   - `protectionInstantiationName`: mandatory, a unique name for the PI;
   - `protectionName`: mandatory, the protection the PI refers to, must be one of the values defined in the XSD;
   - `variableAnnotation`: the annotation that must be written in the source code to force the ACTC to apply the PI to the variables;
   - `codeRegionAnnotation`: the annotation that must be written in the source code to force the ACTC to use the PI on the code region;
   - `attestatorAnnotation`: if the protection uses an attestator, the relative annotation that must be written in the source code, and will be replaced by the ACTC with the attestator in the deploy phase;
   - `verifierAnnotation`: if the protection uses a verifier, the relative annotation that must be written in the source code, and will be replaced by the ACTC with the verifier in the deploy phase;
   - `toolForDeployment`: a list of the tool that must be used by the ACTC to deploy the protection;
   - `clientTimeOverhead`: mandatory, a linear formula that must be used by the ADSS to estimate the client time overhead (see Section 4.2.4 of D5.13); the formula can contain ACTC metrics relative to the asset that must be protected, enclosed between sharp signs (i.e. `#metric#`);
   - `clientMemoryOverhead`: mandatory, as before but to estimate the client memory overhead;
   - `serverTimeOverhead`: mandatory, as before but to estimate the server time overhead;
   - `serverMemoryOverhead`: mandatory, as before but to estimate the server memory overhead;
   - `networkOverhead`: mandatory, as before but to estimate the network overhead;
4. make sure that the created XML file validates against `protectionInstantiation.xsd`;
5. add to the extension `eu.aspire_fp7.adss.akb.protectionInstantiations`, located in the `plugin.xml` of the same package, the path to the created XML file; this can be done from Eclipse by simply:
   (a) opening the `plugin.xml` file with the built-in *Plug-in Manifest Editor*;
   (b) from the *Extension* tab, in the *All Extension* column, right click on the aforementioned 6extension and, from the contextual menu, go in the sub-menu *New* and click on *protectionInstantiationFile*;

(c) click on the created sub-entry, and, from the column *Extension Element Details*, write in the *xmlFile* line the path to the created XML file.

In the next execution, the ADSS will find the added protection instantiations. Then, we provide an example of XML file for a PI.

```
<?xml version="1.0" encoding="UTF-8"?>
<protectionInstantiationsList
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://protectionInstantiation.akb.adss.aspire_fp7.eu"
        xsi:schemaLocation="../schema/protectionInstantiation.xsd">
               <protectionInstantiation>
                       <protectionInstantiationName>Binary Obfuscation (flatten function,
                               low overhead)</protectionInstantiationName>
                       <protectionName>binaryCodeControlFlowObfuscation</protectionName>
                       <codeRegionAnnotation>obfuscations,
                               enable_obfuscation(flatten_function:percent_apply=10)
                               </codeRegionAnnotation>
                       <clientTimeOverhead>   1 + (10 * max(0, #nr_bbls_dynamic_size# /
                               #nr_bbls_static_size# - 10/100) * #nr_bbls_dynamic_size#) /
                               #nr_ins_dynamic_size# </clientTimeOverhead>
                <clientMemoryOverhead>0</clientMemoryOverhead>
                       <serverTimeOverhead>1</serverTimeOverhead>
                <serverMemoryOverhead>0</serverMemoryOverhead>
                <networkOverhead>0</networkOverhead>
               </protectionInstantiation>
               <protectionInstantiation>
                       <protectionInstantiationName>Binary Obfuscation (flatten function,
                       medium overhead)</protectionInstantiationName>
                       <protectionName>binaryCodeControlFlowObfuscation</protectionName>
                       <codeRegionAnnotation>obfuscations,
                               enable_obfuscation(flatten_function:percent_apply=20)
                               </codeRegionAnnotation>
                       <clientTimeOverhead>   1 + (10 * max(0, #nr_bbls_dynamic_size# /
                               #nr_bbls_static_size# - 20/100) * #nr_bbls_dynamic_size#) /
                               #nr_ins_dynamic_size# </clientTimeOverhead>
                <clientMemoryOverhead>0</clientMemoryOverhead>
                       <serverTimeOverhead>1</serverTimeOverhead>
                <serverMemoryOverhead>0</serverMemoryOverhead>
                <networkOverhead>0</networkOverhead>
               </protectionInstantiation>
               <protectionInstantiation>
                       <protectionInstantiationName>Binary Obfuscation (flatten function,
                       high overhead)</protectionInstantiationName>
                       <protectionName>binaryCodeControlFlowObfuscation</protectionName>
                       <codeRegionAnnotation>obfuscations,
                               enable_obfuscation(flatten_function:percent_apply=30)
                               </codeRegionAnnotation>
                       <clientTimeOverhead>   1 + (10 * max(0, #nr_bbls_dynamic_size# /
                               #nr_bbls_static_size# - 30/100) * #nr_bbls_dynamic_size#) /
                               #nr_ins_dynamic_size# </clientTimeOverhead>
                <clientMemoryOverhead>0</clientMemoryOverhead>
                       <serverTimeOverhead>1</serverTimeOverhead>
                <serverMemoryOverhead>0</serverMemoryOverhead>
                <networkOverhead>0</networkOverhead>
               </protectionInstantiation>
    </protectionInstantiationsList>
```

This example XML file contains three PIs for Binary Obfuscation. Since this protection needs only an annotation on the target code region to be applied on the binary by the ACTC, only the `codeRegionAnnotation` element is present. The client time overhead estimation formula uses various ACTC metrics, written enclosed in sharp signs, e.g. `#nr_bbls_dynamic_size#`. When the ADSS consider applying a PI to a target code region, it evaluates the overhead formulas, substituting automatically the sharp sign enclosed ACTC metrics with the actual metric values for the target code region. Also, since the protection has an impact only on the CPU time of the client machine, the other overheads are null: the server time overhead is equal to 1, since it is expressed as a multiplicative factor; instead, the other overheads are equal to 0, since they are expressed as absolute quantities.

15.2.2 Adding new ontologies

To expand the AKB with a new OWL ontology file, the latter must be added to the ADSS by adding the OWL file path in the `eu.aspire_fp7.adss.akb.ontologies` extension of the `plugin.xml` file of the same package, with the same procedure described in Section 15.2.1.

To allow the ADSS to access the entities (and their instances) declared in new ontology files at source code level, for every new entity, a new class must be added in the XCore[12] file `akb.xcore`, contained in the `eu.aspire_fp7.adss.akb` package.

To bind the object and data properties of the new ontology individuals to the attributes of new classes created in the XCore file, an annotation must be added in the Java interfaces generated by the XCore, which are located in the `src-gen` folder of the `eu.aspire_fp7.adss.akb` package. The Java annotation must be called `MapsToIndividual`, and must compulsorily contain the `iri` property set to the IRI of the class containing the individual and the `name` property specifying the individual's name. If the entity contains one or more object properties, the `objectProperties` attribute must be present in the annotation, and it will contain, for every object property, another annotation of the `MapsToObjectProperty` type, with the following attributes:

- `id`: the identifier of the attribute to which the entity must be bind, from the `AkbPackage` class;
- `iri`: the IRI of the object property in the ontology.

Similarly, if the entity contains one or more data property, the `MapsToIndividual` annotation must contain a `dataProperties` attribute, which in turn will contain a `MapsToDataProperty` annotation for every data property of the entity; the latter annotation must contain the attributes already listed for the `MapsToObjectProperty` annotation.

15.2.3 Adding new attack steps

The attack paths detection phase (see Section 14.2) can be extended with new Prolog files containing new attack steps and/or new facts: the paths of the new Prolog files must be added in the `eu.aspire_fp7.adss.akb.prolog` extension of the `plugin.xml` file of the same package, with the same procedure described in Section 15.2.1.

Attack steps must follow the syntax

```
attackStep(<name>(<parameters>),[<premises>],[<conclusions>])) := <conditions>
```

where the data in angle quotes have the following meaning:

- name: an unique name for the attack step;
- parameters: a comma-separated list of Prolog variables; the value assumed by the variables will characterize instances of attack steps in the inferred attack paths (e.g. a variable can be the name of the asset target of the attack);
- premises: a comma-separated list of attack steps that must precede this attack step in any attack path; attack steps must be written following the syntax `<name>(<parameters>)`;
- conclusions: a comma-separated list of facts that stand true if the attack step is correctly completed by the attacker;
- conditions: the body of the rule, a comma-separated list of facts that must stand true for the attacker to succeed in completing correctly the attack step.

[12]https://wiki.eclipse.org/Xcore

Facts must follow the syntax

`fact(<name>(<parameters>)) := <conditions>`

where the data in angle quotes have the following meaning:

- name: an unique name for the attack step;
- parameters: a comma-separated list of Prolog variables; the value assumed by the variables will characterize the instances of the facts in the attack paths detection phase;
- conditions: the body of the rule, a comma-separated list of facts that must stand true for this fact to be true.

Facts may be contained in the conditions of other facts or attack steps, following the syntax `fact(<name>(<parameters>))`. At the beginning of the attack paths detection phase, the ADSS automatically creates the instances of the following list of facts from data in the AKB:

- `code(<name>)`: instantiated for every code region in the AKB; `<name>` is the name of the code region;
- `datum(<name>)`: instantiated for every datum in the AKB; `<name>` is the name of the code region;
- `hasProperty(<name>,<property>)`: instantiated for every asset in the AKB; `<name>` is the name of the asset, `<property>` is the name of the security property that the asset have;
- `contains(<container>,<containee>)`: instantiated for every application part containing other application parts; the parameters are respectively the names of the container and the containee application part;
- `isContainedBy(<containee>,<container>)`: the opposite of the `contains` fact;
- `accesses(<code>,<datum>)`: instantiated for every code region that access to a datum; the parameters are their names;
- `isAccessedBy(<datum>,<code>)`: the opposite of the `accesses` fact;
- `calls(<caller>,<callee>)`: instantiated for every code region that contain a call to another code region; the parameters are the name of the caller and callee code regions;
- `isCalledBy(<callee>,<caller>)`: the opposite of the `accesses` fact.

Facts of this kind can be contained in the conditions of attack steps and facts using the syntax `<fact_name>(<parameters>)`, without enclosing them in the `fact` composite term (differently from user defined facts).

We provide an example of attack steps and facts KB. Note that in Prolog lines starting with the percentage sign are comments.

```
% An asset can be a code or a datum, and can belong to the attacker copy or
% to the victim copy.
fact(asset(Asset)) :-
(code(Asset); datum(Asset));

% To breach the confidentiality of an asset we must know its content from
% the victim.
fact(breached(Asset, confidentiality)) :-
          hasProperty(Asset, confidentiality),
          fact(asset(Asset)),
          fact(contentRetrieved(Asset)).
```

```
% If a part was statically or dynamically retrieved, well, it was retrieved.
fact(contentRetrieved(Part)):-
           fact(asset(Part)),
           (fact(contentStaticallyRetrieved(Part));
           fact(contentDynamicallyRetrieved(Part))).

% If I know the static location of an hardcoded part I know its content.
fact(contentStaticallyRetrieved(Part)) :-
           (
                   fact(asset(Part)),
                   hasProperty(Part, hardcoded))),
                   fact(staticallyLocated(Part))
           ).

% The attacker can statically locate a hardcoded part in its code.
attackStep(staticallyLocate(Part), [], [staticallyLocated(Part)]) :-
 (code(Asset);
 (datum(Asset), hasProperty(Asset, hardcoded))).
```

In this example we present how the ADSS infers the attack paths starting from attack steps and facts, and how the latter are written in form of Prolog rules. Suppose we have an asset named `a1`, which is a hard-coded datum and has the confidentiality security property. The ADSS will query the Prolog-based KB, asking for the attack paths that end breaching the confidentiality of `a1`. The Prolog engine will therefore perform the backward reasoning process, trying to validate the conditions of `fact(breached(a1, confidentiality))` (obtained assigning the value `a1` to the variable `Asset`). The first condition `hasProperty(a1, confidentiality)` is true because the ADSS has *asserted* this information in the Prolog-based KB at the start of the attack paths detection phase. The second condition is `fact(asset(a1))`, which means that the Prolog engine must evaluate the corresponding rule, which says that `a1` is an asset if it is a code region or a datum, and is therefore validated, since the ADSS has asserted in the Prolog-based KB, at the start of the attack paths detection phase, that `a1` is a datum. The third condition is `fact(contentRetrieved(a1))`, which leads the Prolog engine to validate the relative rule, which states that a content can be retrieved in a static or dynamic way: the Prolog engine will try both ways, but we report only the static way for the sake of brevity. This leads to the verification of `fact(contentStaticallyRetrieved(a1))`, that has as a condition `fact(staticallyLocated(a1))`, which in turn is contained in the conclusions of `attackStep(staticallyLocate(a1))`. So the Prolog engine will conclude that a way to breach the confidentiality of asset `a1` is the attack path constituted by a single attack step, `attackStep(staticallyLocate(a1))`.

Now suppose we add the following rules to the KB

```
fact(breached(Asset, integrity)) :-
        hasProperty(Asset, integrity),
        fact(asset(Asset)),
        fact(changed(Asset)).

% If a part was statically or dynamically changed, well, it was changed.
fact(changed(Part)) :-
        fact(asset(Part)),
        (fact(staticallyChanged(Part)); fact(dynamicallyChanged(Part))).

attackStep(staticallyChange(Part), [staticallyLocated(Part)],
[staticallyChanged(Part)]) :-
        fact(asset(Part)),
        hasProperty(Part, hardcoded))).
```

and there is a hard-coded asset `a2`, which has the integrity security property. Similarly as before, the ADSS will query the Prolog engine for the attack paths that breach the integrity of `a2`, which translate in the validation of `fact(breached(a2, integrity))`. This leads in turn to the validation of `fact(changed(a2))`: this rule says that an asset may be changed either statically or

dynamically; again, we report only for the static way. The rule body leads to the validation of `fact(staticallyChanged(a2))`, conclusion of `attackStep(staticallyChange(a2))`. But this attack step has a premise, `staticallyLocated(a2))`, which in turn is the conclusion, as we seen in the previous example, of `attackStep(staticallyLocate(a2))`. In the same way of the previous example, the Prolog engine concludes that this attack step is feasible for the attacker without needing the completion of other attack steps, and therefore the query evaluation stops, giving as result that the attack path composed by the following attack steps (in the attacker execution order)

1. `attackStep(staticallyLocate(a2))`
2. `attackStep(staticallyChange(a2))`

leads to the breaching of the `a2` asset integrity property. The fact that this attack path leads also to the breaching of the `a2` asset confidentiality property is also detected and reported.

15.2.4 Adding a new solver for the L2P MILP problem

To support a new MILP solver in the second level protection discovery phase (see Section 14.5), a new implementation of the `eu.aspire_fp7.adss.optimizationAPI.Optimizer` Java interface must be added to the latter. This interface serves as an abstraction layer of API provided by MILP solvers. Description of methods that must be implemented can be found in the Javadoc of the interface located at `doc/eu/aspire_fp7/adss/optimizationAPI/Optimizer.html` in the `eu.aspire_fp7.adss` package.

15.3 API

The ADSS is modeled via a model-view-controller paradigm so that its core functionalities are independent from its user interface. This allows a developer to fully use the inferential capabilities of the ADSS programmatically in its own application or build a new UI from scratch with ease.

The following sections list the most important classes and methods of the ADSS API.

15.3.1 eu.aspire_fp7.adss.akb.AKBUtil

The `eu.aspire_fp7.adss.akb.AKBUtil` class is a utility class that can be used to manage the AKB and it offers the following methods:

static public `Model loadFromOntology(IFile file)`
: Converts a file containing an OWL ontology into the AKB model (see Section 15.3.3).

static public void `saveIntoOntology(Model model)`
: Saves the AKB internal model into its related OWL ontology object. Note that you need to call the `model.getOntology().save()` method to actually save the ontology to a file.

static public `Ontology createEmptyOntology(Preferences preferences)`
: Creates an empty AKB with some initial preferences. Note that you need to call the `ontology.save()` method to actually save the ontology to a file.

15.3.2 eu.aspire_fp7.adss.ADSS

The `eu.aspire_fp7.adss.ADSS` class is the main connection point to access the ADSS functionalities. Its public methods are:

`public ADSS(IFile file, Model model)`
Creates an ADSS instance related to an OWL file and an AKB model. Use the method `AKBUtil.loadFromOntology()` to get the model from the file.

`public void analyzeSourceCode()`
Executes the source code analysis phase (see Section 14.1). The results are stored in the model.

`public void findAttacks()`
Executes the attack paths detection phase (see Section 14.2). The results are stored in the model.

`public void findProtections()`
Executes the protections detection phase (see Section 14.3). The results are stored in the model.

`public void findFirstLevelProtections()`
Executes the first level protections discovery phase (see Section 14.4). The results are stored in the model.

`public void findSecondLevelProtections(Solution solution)`
Executes the second level protections discovery phase (see Section 14.5) for a specific solution. The results are stored in the model.

`public void deploySolution(Solution solution)`
Executes the solution deployment phase (see Section 14.6 by deploying the chosen solution.

`public void save()`
Saves the AKB model to an OWL file, also creating a backup of the old AKB.

15.3.3 eu.aspire_fp7.adss.akb.Model

The AKB has actually two representations in the ADSS memory: as an ontology and as a set of Java objects. The `eu.aspire_fp7.adss.akb.Model` represents the entry point that allows a developer to access both. The ontology is a sort of low-level representation of the AKB, while the set of Java objects are its high-level representation and they are the advised way to access and change the AKB. If these objects are modified, the corresponding ontology must be synchronized via the `eu.aspire_fp7.adss.akb.AKBUtil.saveIntoOntology()` method.

The most important methods of the `eu.aspire_fp7.adss.akb.Model` are:

`public Ontology getOntology()`
Retrieves the ontology related to the AKB.

`public void setOntology(Ontology ontology)`
Sets the ontology of the AKB.

`public EList<ApplicationPart> getApplicationParts()`
Retrieves the list of application parts.

`public EList<AttackPath> getAttackPaths()`
Retrieves the list of attack paths.

`public EList<AttackStep> getAttackSteps()`
Retrieves the list of attack steps.

`public Preferences getPreferences()`
Retrieves the preferences of the ADSS.

`public void setPreferences(Preferences preferences)`
Sets the preferences of the ADSS.

`public EList<Rule> getRules()`
Retrieves the list of user-defined rules.

`public EList<Protection> getProtections()`
Retrieves the list of protections.

public `EList<ProtectionObjective> getProtectionObjectives()`
Retrieves the list of protection objectives.

public `EList<AppliedProtectionInstantiation> getAppliedPIs()`
Retrieves the list of applied protection instantiations.

public `EList<ApplicationPart> getAssets()`
Retrieves the list of assets.

public `Attacker getAttacker()`
Retrieves the attacker profile.

public void `setAttacker(Attacker attacker)`
Sets the attacker profile.

public `EList<AttackerTool> getAttackerTools()`
Retrieves the list of attacker tools.

public `Solution getVanillaSolution()`
Retrieves the solution related to the vanilla application.

public void `setVanillaSolution(Solution value)`
Sets the solution related to the vanilla application.

public `EList<Solution> getSolutions()`
Retrieves the list of L1P and L2P solutions.

16 Conclusions

This document documents the state of the ACTC and the ADSS at the end of the ASPIRE project, as they are delivered as D5.10 of type prototype. The foreseen protection techniques have been integrated into the ACTC, which has been coupled to the ADSS. Based on protection annotations, the ACTC can protect applications. Based on requirement annotations and additional user input, the ADSS can invoke the ACTC to determine the impact of protections, and then select an optimized combination of protections.

Most of the prototypes discussed and documented in this document are open-sourced as D5.12, as documented further in D5.13.

Part IV

Appendices

A Data-Specific Annotations

Section authors:
Roberto Tiella (FBK), Jerome d'Annoville (GTO)

Data annotations concern protection of variables. Both variable declarations and structure fields can be annotated.

Semantics: The current implementation of data hiding obfuscation is not inter-procedural: encoded variables are decoded before being used as actual parameters and return parameters are decoded on exiting the function.

A.1 XOR (1-1 encoding)

```
<PROTECTION_NAME> ::= xor

<PROTECTION_PARAMETER> ::=   mask(constant(<INTEGER>))
                           | mask(random(<INTEGER>,<INTEGER>)
                           | mask(dynamic)
                           | [opaque(clique, <INTEGER>, <INTEGER>)]
```

Semantics: Requires the tool to encode the annotated variable using XOR-encoding as described in deliverables D2.01 and D2.08.

- `mask(constant(<INTEGER>))` : using the mask number specified;
- `mask(random(<INTEGER>,<INTEGER>))` : using a random mask number in the range specified;
- `mask(dynamic)` :(optionally) using a random mask that is dynamically generated at runtime;
- `opaque(clique, <INTEGER>, <INTEGER>)` : (optionally) the mask parameter is encoded using a k-clique opaque constant derived by a 3SAT problem with a number of propositional variables specified by the first integer. The constant is encoded using a number of bits specified by the second integer.

Only one `mask` parameter and optionally one `opaque` parameter can be specified. `mask(dynamic)` and `opaque` can not be both present in the same annotation.

Examples: To require the variable `x` to be encoded using XOR with 12 as a mask:

```
int x __attribute__((ASPIRE("protection(xor,mask(constant(12)))"))) = 28 ;
```

To require the variable `x` to be encoded using XOR with a random opaque mask:

```
int x __attribute__((ASPIRE("protection(xor,mask(random(1,255),
      opaque(clique,4,16)))"))) = 28 ;
```

the mask is chosen between 1 and 255 and it is encoded using 16 bits. Furthermore the constant is opacified using k-problems derived from 3SAT formulas in 4 propositional variables.

A.2 Merge Scalar Variables

Scalar variables can be merged as described in deliverable D2.01.

```
<PROTECTION_NAME> ::= merge_vars

<PROTECTION_PARAMETER> ::=   set(<ID>)
                           | coord(<INTEGER>)
                           | size(<INTEGRAL_SIZE>)
                           | offset(<INTEGER>)
```

All the above parameters are mandatory.

Semantics:

- `set(<ID>)`: the name of the set the variable belongs to. Variable belonging to the same set are packed into a single memory area;
- `coord(<INTEGER>)`: the position, starting from 1, of the variables in the packed memory area;
- `size(<INTEGRAL_SIZE>)`: the number of bits allocated to the variable;
- `offset(<INTEGER>)`: offset to be added to variable's value.

Example: To specify to pack `x` and `y` variables in a byte using 6 bits for `x` and 2 bits for `y`:

```
int x __attribute__((ASPIRE("protection(merge_vars,set(s1),coord(1),
                                        size(6), offset(32))"))) = 0;

int y __attribute__((ASPIRE("protection(merge_vars,set(s1),coord(1),
                                        size(2), offset(0))"))) = 0;
```

Furthermore for variables x an offset of 32 will be added to its value to manage negative values.

A.3 Residue Number Coding

Values can be encoded using residue number coding (RNC) as described in deliverables D2.01 and D2.08.

```
<PROTECTION_NAME> ::= rnc

<PROTECTION_PARAMETER> ::=   base(constant(<LIST_OF_INTEGERS>))
                           | base(random(<INTEGER>,<INTEGER>))
                           | [ opaque(clique,<INTEGER>,<INTEGER>) ]
```

Semantics:

- `base(constant(<LIST_OF_INTEGERS>))`: specifies the sequence of pairwise prime integers used in the encoding. In the current implementation, two numbers can be specified.
- `base(random(<INTEGER>,<INTEGER>))`: specifies that the sequence of pairwise prime integers used in the encoding has to be chosen randomly in the specified range.
- `opaque(clique, <INTEGER>, <INTEGER>)` : (optionally) RNC parameters are encoded using k-clique opaque constants derived by 3SAT problems with a number of propositional variables specified by the first integer. Constant are encoded using a number of bits specified by the second integer.

Only one `base` parameter and optionally one `opaque` parameter can be specified.

Examples: Encode variables `x`, `y` and `z` using RNC with constants 31 and 29:

```
int x __attribute__((ASPIRE("protection(rnc,base(constant(31,29)))")));
int y __attribute__((ASPIRE("protection(rnc,base(constant(31,29)))")));
int z __attribute__((ASPIRE("protection(rnc,base(constant(31,29)))")));
```

Encode variables `a` and `b` using RNC with random parameters chosen between 100 and 200:

```
int a __attribute__((ASPIRE("protection(rnc,base(random(100,200)))")));
int b __attribute__((ASPIRE("protection(rnc,base(random(100,200)))")));
```

Encode variable `u` using RNC with parameters encoded as opaque constants:

```
int u __attribute__((ASPIRE("protection(rnc,base(constant(31,29)),
                             opaque(clique,4,8))")));
```

A.4 Convert Static to Procedural Data

Static data, such as constant strings, can be hidden from static analysis by replacing them by code that generates the data on the fly, as described in deliverable D2.01.

```
<PROTECTION_NAME> ::= data_to_proc

<PROTECTION_PARAMETER> ::=   algorithm (mealy_lutable | mealy_switch)
```

Semantics:

- `algorithm(mealy_lutable)`: transformed code is based on a Mealy machine implemented using lookup tables.
- `algorithm(mealy_switch)`: transformed code is based on a Mealy machine implemented using switch statements.

Example: Encode a string as a procedure using a look-up table based Mealy machine:

```
const char * key __attribute__((
  ASPIRE("protection(data_to_proc,algorithm(mealy_lutable))")
  )) = "password";
```

A.5 Multi-threaded Cryptography

As described in Section 3.6 of deliverable D1.04, multi-threading is used to protect cryptographic operations. The immediate key value must be specified on the key variable declaration as specified below. Plain text and cipher text variables are specified on the code annotation described in Section B.3.

```
<PROTECTION_NAME> ::=   multi_threaded_crypto

<PROTECTION_PARAMETER> ::=   algorithm (<SYMMETRIC_KEY_ALGORITHM>)
                         |   mode(<MODE>)
                         |   key (<KEY_VALUE>)

<SYMMETRIC_KEY_ALGORITHM> ::=  AES

<MODE> ::= CBC

<KEY_VALUE> ::= <SQ_STRING>
```

Semantics: Algorithm, mode and the key protection parameter must be specified.

- `SYMMETRIC_KEY_ALGORITHM and MODE` : only AES in CBC mode is supported by the protection so far.
- `KEY_VALUE` : Key value specified in Base64

Example: Specify an AES key value:

```
1   char * key __attribute__((ASPIRE("protection(multi_threaded_crypto,"
2                                    "algorithm(AES),"
3                                    "mode(CBC),"
4                                    "key('MDEyMzQ1Njc4OUFCQ0RFRg==')")));
```

An more extensive example with both data annotations and the corresponding code annotations can be found in Section B.3.

A.6 Software Time Bombs

The protection of time bombs is described in Section 4.7 of deliverable D1.04. Its code annotations are specified in Section B.12, its corresponding data annotations are the following:

```
<PROTECTION_NAME> ::= timebombs

<PROTECTION_PARAMETER> ::= code_area_candidate (<LIST_OF_IDs>)
```

Semantics:

- Specifies that the variable is a good candidate for memory corruption in case software time bombs have to be triggered because of a detected attack. The variable scope should be global, so that any function can refer to it.
- `code_area_candidate` Enables linking this variable to portion(s) of the code where the application developer would like the time bombs to be invoked. The ID(s) listed here should match with those specified on ASPIRE begin code annotation IDs for the time bombs protection name method specified in Section B.12.

Example: The handle on a structure frequently used in the application is targeted for the software time bombs mechanism:

```
struct struct_name *handle __attribute__((
  ASPIRE("protection(timebombs,"
         "code_area_candidate(protocol_manager_function))")
 ));

int protocol_manager(int x, int y) {
  _Pragma("ASPIRE begin protection(timebombs,"
          "code_area(protocol_manager_function))");
  // ... function code
  _Pragma("ASPIRE end");
}

int main() {
  _Pragma("ASPIRE begin protection(timebombs, init);
  _Pragma("ASPIRE end");
  // ... continue
}
```

A.7 Diversified Cryptographic Library

The DCL protection is described thoroughly in Section 6 of deliverable D2.10. Three crypto operation are available as explained below. The annotation has to paired with integer variable declaration, because it will holds the result of execution.

With DCL protection, one can do key derivation. The advantage of using this protection is that the master key is completely hidden. Input is the object to be derived, the salt for hashing, and the amount of iteration. The algorithm used is PBKDF2 HMAC SHA1.

```
<PROTECTION_NAME> ::= dcl

<PROTECTION_PARAMETER> ::= kdf(<KDF_PARAMETER>)

<KDF_PARAMETER> ::= <INPUT> | <INPUT_LEN> | <SALT> | <SALT_LEN>
                  | <ITERATION> | <OUT> | <OUT_LEN>
```

Semantics:

- The object to be derived is provided with `INPUT`, and `INPUT_LEN` as its length.
- To strengthen the hash function, it is required to provide a salt. This value shall be put as `SALT`, and its length `SALT_LEN`.
- Value of `ITERATION` refer to the amount of hash repetition.

- The derived key object will be put in `OUT`, and the length in `OUT_LEN`.
- If the operation is success, the annotated variable will have value of 0.

Example: The following example shows snippet code to use key derivation on DCL:

```
int perform_KDF(mbyte * out,
  mbyte * master_key, size_t size_master_key,
  mbyte * imei, size_t size_imei,
  mbyte * random_value, size_t size_random_value,
  mbyte * salt_key, size_t size_salt_key) {

  mbyte ** args = (mbyte **) malloc (2*sizeof(mbyte *));
  size_t size_password = size_imei+size_random_value;
  size_t outSize = KEK_LEN_BYTE;
  int iteration = KEK_ITERATION;
  *args = imei;
  *(args + 1) = random_value;
  concat(args, &password, 2, size_imei, size_random_value);

  int result __attribute__((ASPIRE("protection(dcl, kdf(
    password, password_len, salt_key, size_salt_key,
    iteration, out, &out_size))")));
  free(password);
  free(args);

  return result;
}
```

Another DCL operation available is to generate One-Time Password. The input is a key and input message, and it will generate an OTP as the output, using HMAC SHA1 algorithm.

```
    <PROTECTION_NAME> ::= dcl

    <PROTECTION_PARAMETER> ::= HOTP(<HOTP_PARAMETER>)

    <HOTP_PARAMETER> ::= <KEY> | <KEY_LEN> | <INPUT> | <INPUT_LEN>
                       | <OTP_LEN> | <OUT> | <OUT_LEN>
```

Semantics:

- The key of hash function is provided with `KEY`, and `KEY_LEN` as its length.
- The object to be input of the hash is provided with `INPUT`, and length `INPUT_LEN`.
- The desired length of a OTP is provided in `OTP_LEN`.
- The generated OTP object will be put in `OUT`, and the length in `OUT_LEN`.
- If the operation is success, the annotated variable will have value of 0.

Example: The following example shows snippet code to use OTP generation on DCL:

```
int generate_otp(otp_stored_attributes *otp_stored_infos,
  mbyte *otp, size_t size_otp) {

  mbyte *ivec = (mbyte *) malloc (16 * sizeof (mbyte));
```

```
  memcpy (ivec, iv, 16);
  size_t plaintext_len = ciphertext_len;
  uint8_t* output = malloc (sizeof(uint8_t) * plaintext_len);

  int result __attribute__((ASPIRE("protection(dcl, HOTP(
  otp_stored_infos->device_key,
  otp_stored_infos->size_device_key,
  otp_stored_infos->counter_value_byte,
  otp_stored_infos->size_counter_value_byte,
  size_otp, otp, &size_otp))")));
  return result;
}
```

The third operation is AES decryption with CBC block. All parameters have to be supplied, i.e. the key, ciphertext, and the placeholder to contains the output. After data declaration finished, the variable will have value 0 if the operation is successful.

```
    <PROTECTION_NAME> ::= dcl

    <PROTECTION_PARAMETER> ::= AES_CBC_DEC(<AES_PARAMETER>)

    <AES_PARAMETER> ::= <KEY> | <KEY_LEN> | <IV> | <INPUT> | <INPUT_LEN>
                      | <OUT> | <OUT_LEN>
```

Semantics:

- The key for decryption is provided with `KEY`, and `KEY_LEN` as its length.
- The initial vector is provided with `IV`.
- The ciphertext to be decrypted is provided with `INPUT`, and length `INPUT_LEN`.
- The resulted plaintext will be put in `OUT`, and the length in `OUT_LEN`.
- If the operation is success, the annotated variable will have value of 0.

Example: The following example shows snippet code how to use AES decryption on DCL:

```
size_t aes_decrypt(uint8_t *key, size_t  key_len, uint8_t *ciphertext,
  size_t ciphertext_len, uint8_t *iv, uint8_t **plaintext) {

  mbyte *ivec = (mbyte *) malloc (16 * sizeof (mbyte));
  memcpy (ivec, iv, 16);
  size_t plaintext_len = ciphertext_len;
  uint8_t* output = malloc (sizeof(uint8_t) * plaintext_len)

  int result __attribute__((ASPIRE("protection(dcl, AES_CBC_DEC(
    key, key_len, ivec, ciphertext, ciphertext_len,
    output, &plaintext_len))")));
  if (result != 0)
    result = -1;
  else
    result =  plaintext_len - 16;
  return result;
}
```

B Code-Specific Annotations

Section authors:
Bjorn De Sutter, Bart Coppens (UGent), Rachid Ouchary, Alessio Viticchié, Cataldo Basile (POLITO), Brecht Wyseur (NAGRA), Roberto Tiella (FBK), Alessandro Cabutto (UEL), Werner Dondl (SFNT), Jerome d'Annoville (GTO)

B.1 White-box Cryptography

The white-box cryptography (WBC) protections are described in Section 3.5 of deliverable D1.04.

```
<PROTECTION_NAME> ::= wbc

<PROTECTION_PARAMETER> ::=   label       ( <ID> )
                           | role        ( key | input | output | iv)
                           | size        ( <INTEGER> )
                           | algorithm   ( aes | des | tdes | rsapub | rsapriv )
                           | mode        ( ECB | CBC | CBC_INV )
                           | operation   ( encrypt | decrypt )
```

Semantics:

- `label`: an internal identifier of the crypto operation to protect using WBC; the same label must be used for all elements related to a WBC transformation; mandatory.
- `role`: role of the data; only apply to data; may be key (initialized with its value for a fixed-key algorithm),input or/and output; mandatory for data.
- `size`: size of the data, in bytes; mandatory for data.
- `algorithm`: algorithm to use; mandatory in the Pragma.
- `mode`: chaining mode to use; default to ECB.
- `operation`: operation to use; default to decrypt.

Example 1: Fixed-key AES decryption, ECB mode

```
static const char ciphertext[] __attribute__
  ((ASPIRE("protection(wbc,label(ExampleFixed),role(input),size(16))")))
  = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
      0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };

static const char key[] __attribute__
  ((ASPIRE("protection(wbc,label(ExampleFixed),role(key),size(16))")))
  = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
      0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff };

char plaintext[16] __attribute__
  ((ASPIRE("protection(wbc,label(ExampleFixed),role(output),size(16))")));

_Pragma ("ASPIRE begin protection(wbc,label(ExampleFixed),algorithm(aes),"
         "mode(ECB),operation(decrypt))")
decrypt_aes_128(ciphertext, plaintext, key);
_Pragma("ASPIRE end");
```

In this example, the call to encrypt_aes_128 must be replaced by a call to the WBC function. The next frame depicts the resulting code, after transformation. The NULL parameter corresponds to the initialization vector, not used in ECB mode. The name of the header file to include, as well as the name of the generated source files, does not appear in the annotations; it is managed by the ACTC.

```
#include "wbc_example_fixed.h"

static const char ciphertext[]
  = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07
      0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };

char plaintext[16];

wbgcClientDecryptExampleFixed(ciphertext, 16, NULL, plaintext);
```

Example 2: Dynamic-key DES encryption, CBC mode

```
static const char init_vector[] __attribute__
  ((ASPIRE("protection(wbc,label(ExampleDynamic),role(iv),size(8))")))
  = {0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7};

char plaintext[64] __attribute__
  ((ASPIRE("protection(wbc,label(ExampleDynamic),role(input),size(64))")));

char* key __attribute__
  ((ASPIRE("protection(wbc,label(ExampleDynamic),role(key),size(8))")));

char ciphertext[64] __attribute__
  ((ASPIRE("protection(wbc,label(ExampleDynamic),role(output),size(64))")));

init_plain_text(plaintext);
init_key(key);

_Pragma ("ASPIRE begin protection(wbc,label(ExampleDynamic),algorithm(des),"
         "mode(CBC),operation(encrypt))")
encrypt_des_cbc(plaintext, 64, ciphertext, 64, init_vector, key);
_Pragma("ASPIRE end");
```

In this example, the call to encrypt_des_cbc must be replaced by a call to the WBC function. The next frame depicts the resulting code, after transformation. The name of the header file to include, as well as the name of the generated source files, does not appear in the annotations; it is managed by the ACTC.

```
#include "wbc_example_dynamic_clt.h"

char plaintext[64];

char* key;

char ciphertext[64];

init_plain_text(plaintext);
init_key(key);

wbgcClientEncryptExampleDynamic(plaintext,64,key,8,ciphertext);
```

B.2 Client-Side Code Splitting by means of SoftVM

The client-side code splitting protections are described in Section 3.5 of deliverable D1.04. The corresponding annotations are as follows:

```
<PROTECTION_NAME> ::= softvm

<PROTECTION_PARAMETER> ::= ( softvm | application ) ( mobile )?
```

Semantics:

- Case `(softvm)`: Marks the annotated code region for possible SoftVM protection, i.e., for translation to bytecode to be interpreted by the SoftVM. This means the protection tools will search in the annotated code region for slices that can be translated and run in the SoftVM.
- Case `(application)`: Marks the annotated code region as code that should remain native, i.e., that will not be translated into bytecode for the SoftVM. This prohibition is absolute: even if the ADSS would make decisions that move additional code to the SoftVM, regions marked with `(application)` will *never* be moved to the SoftVM.
- Case `(mobile)`: Marks the SoftVM protected code region as to be made mobile. Only effective in combination with the `(softvm)` annotation.
- The `softvm` annotation does not propagate across function calls.

```
1   void f(int x) {
2     return x;
3   }
4   void g(int x) {
5     _Pragma("ASPIRE begin softvm(softvm)");
6     int z = (x + x) ^ 2;
7     z = z * x;
8     z = f(z);
9     _Pragma("ASPIRE end");

10    int bound = x+1;

11    _Pragma("ASPIRE begin softvm(application)");
12    for (int i = 0; i <bound; i++)
13      z+=x;
14    _Pragma("ASPIRE end");

15    _Pragma("ASPIRE begin softvm(softvm,mobile)");
16    for (int i = 0; i <bound; i++)
17      z+=x;
18    _Pragma("ASPIRE end");

19    return z;
20  }
```

Example: The binary code associated with lines 6 to 8 is analyzed by the tool flow to determine which instructions can be executed by the SoftVM. For example, suppose that the code on lines 6 and 7 would be compiled into an ADD, an EOR and a MUL instruction, and suppose that the SoftVM would support ADD and EOR instructions, but not MUL instructions. Then only the ADD and EOR would be moved to the SoftVM. Depending on the generated code and the instructions supported by the SoftVM, the assignment of the local variable `z` to the return value of `f` on line 8

is also executed in the SoftVM. However, whether or not the function `f` that is called in line 7 is executed in the application or SoftVM will be decided by the ADSS. The loop on lines 12 and 13 is always executed in the application, regardless of which further decisions the ADSS takes. Finally, the loop on lines 16 and 17 will always be executed in the SoftVM, using mobile bytecode.

B.3 Multi-threaded Cryptography

As described in Section 3.6 of deliverable D1.04, we will use multi-threading to protect cryptographic operations. In this section, we specify are the corresponding code fragment annotations. These code annotations suppose that a corresponding data annotation attribute as described in Section A.5 is specified on the encryption key variable declaration.

```
<PROTECTION_NAME> ::= multi_threaded_crypto

<PROTECTION_PARAMETER> ::=
        symmetric_encrypt (<THREAD_NB>, <KEY>, <PLAINTEXT>, <CIPHERTEXT>)

<THREAD_NB> ::= <INTEGER>

<KEY> ::= <ID>

<PLAINTEXT> ::= <ID>

<CIPHERTEXT> ::= <ID>
```

Semantics:

- `THREAD_NB`: Specifies the maximal number of threads used to protect the process.
- `KEY`: Denotes an identifier that is the symmetric key to be used for the encryption.
- `PLAINTEXT`: Denotes an identifier that is the data to encrypt.
- `CIPHERTEXT`: Denotes an identifier that is the ciphered data.

```
char * key __attribute__((ASPIRE("protection(multi_threaded_crypto,"
                                  "algorithm(AES), mode(CBC),"
                                  "key('VEhJU0lTT05FQUVTS0VZOQ==')")")));
void g(char *data, int datalen) {
  _Pragma("ASPIRE begin protection("multi_threaded_crypto,"
                       "symmetric_encryption(5, key, data, result))");
  AES_encrypt(key, data, datalen, result); // to be replaced
  _Pragma("ASPIRE end");
}
```

Example: In this example the original AES_encrypt call will be replaced by a call to a library that is included. This library will call the crypto server and starts 5 threads, each one using a different key and switching block data and keys at each round. According to the key length (128, 192, 256) there are either 10, 12 or 14 rounds with AES.

B.4 Anti-Debugging

The ASPIRE anti-debugging protections are described in Section 3.2 of deliverable D1.04. The corresponding annotations are as follows:

```
<PROTECTION_NAME> ::= anti_debugging

<PROTECTION_PARAMETER> ::= in                    ( debugger | application )
```

Semantics:

- Case `in(debugger)`: Instruct the anti-debugging protection to execute the annotated code in the debugger.
- Case `in(application)`: Instruct the anti-debugging protection to execute the annotated code in the application, rather than in the debugger. This prohibition is absolute: even if the ADSS would make decisions that move additional code to the debugger, regions marked with `in (application)` will *never* be moved to the debugger.
- The `anti_debugging` annotation does not propagate across function calls.

```
1   void f(int x) {
2     return x;
3   }

4   void g(int x) {
5     _Pragma("ASPIRE begin anti_debugging(in_debugger)");
6     int z = x + x;
7     z = f(z);
8     _Pragma("ASPIRE end");

9     int bound = x+1;

10    _Pragma("ASPIRE begin anti_debugging(in_application)");
11    for (int i = 0; i <bound; i++)
12      z+=x;
13    _Pragma("ASPIRE end");

14    return z;
15  }
```

Example: In this example, the binary code associated with lines 6 and 7 is analyzed to determine which instructions (if any) are supported to be moved to the debugger context. Instructions that cannot be moved to the debugger context will be executed in application context, instructions that can be moved to the debugger context will be moved to the debugger context. If the developer has not specified to execute the code in the debugger or in the application (as is the case here for function `f`), the ADSS will decide and add an annotation with its decision. Similarly, whether or not the assignment on line 9 is executed in the debugger is decided and annotated by the ADSS. The loop on lines 11 and 12 is executed in the application.

B.5 Call-stack Checks

As described in D1.04, the precise form of the call-stack checks that will be integrated in the ASPIRE ACTC has not yet been decided. As the annotations are coupled tightly to the specific form, we cannot provide a specification for this protection's annotations at this point in time.

B.6 Code Guards

The ASPIRE code guard protections are described in Section 4.2 of deliverable D1.04. For this protection, we foresee multiple annotations, defined as follows:

```
<PROTECTION_NAME> ::= guarded_region

<PROTECTION_PARAMETER> ::=  label ( <ID> )
                        |   guarded ( never )

<PROTECTION_NAME> ::= guard_attestator

<PROTECTION_PARAMETER> ::=
           | label ( <ID> )
           | regions (<LIST_OF_IDS>)
           | attest ( never )

<PROTECTION_NAME> ::= guard_verifier

<PROTECTION_PARAMETER> ::=  attestator ( <ID> )
```

Semantics: We provide three different protection annotations, each with their own parameters and semantics:

- The `guarded_region` annotation is used to give more information about which code regions need to be guarded. It can be used to inform the protection technique that the annotated code region must be protected with code guards. In that case, the `label` parameter is required, and is to identify this code region with a unique label.

  This annotation can also be used to specify that the code region may not be verified, in which case the `guarded(never)` parameter needs to be provided, and the `label` annotation shall not be provided.

- The `guard_attestator` annotation is used to inform the protection technique about inserting code guards attestators. The annotation can be used to instruct that the annotated code region cannot contain any attestators, in which case the parameter `attest(never)` should be specified.

  This annotation can be used to specify that the annotated code region must contain a code guard attestator. This is done by specifying both the `label` and `regions` parameters. The `regions` parameter contains a list of code region labels (as they have been labeled by the `label` parameter of `guarded_region` annotations). As with the `guarded_region` annotation, the `label` parameter is used to provide a unique label to each specific attestator.

  If the annotated code region is empty, this instructs the tool chain to insert the attestator at that specific code location.

- The `guard_verifier` annotation is used to specify that a verifier should be inserted in the annotated code region. The `attestator` parameter is required, and is the `label` of a `guard_attestator` of which the generated attestation (i.e., hash) needs to be verified.

  If the region is empty, this instructs the tool chain to insert the verifier at that specific code location.

Example: In this example, the developers know that the `render_frame` function will always be called after the `decode_frame` function, and that the protection technique needs to guard the code of the `decrypt_stream` function:

```
1   void decrypt_stream(Stream* s) {
2     _Pragma("ASPIRE begin protection(guarded_region,label(decryption))");
3     /* Code to decrypt data */
4     _Pragma("ASPIRE end");
5   }

6   void decode_frame(Frame* f) {
7     _Pragma("ASPIRE begin protection(guard_attestator,"
8             "label(decryption_hash), regions(decryption))");
9     /* Code to decode a video frame */
10    _Pragma("ASPIRE end");
11  }

12   void render_frame(Frame* f) {
13     _Pragma("ASPIRE begin protection(guard_verifier,"
             "attestator(decryption_hash))");
14     _Pragma("ASPIRE end");
15     /* Code to render a frame */
16   }
```

The developer assigns the code in the `decrypt_stream` function the "decryption" label for use by the code guards with the Pragma on line 2.

The `guard_attestator` annotation on line 7 instructs the protection technique to insert an attestator labeled "decryption_hash" somewhere in the code region on lines 8-10. This attestator will attest the code regions that are labeled "decryption", i.e., the code between lines 2 and 4.

The empty annotation region on line 13 instructs the protection technique to insert a verifier for the "decryption_hash" attestator that has been inserted in the code region on lines 8-10. Because this is an empty annotation region, the verifier is inserted at this specific code point, i.e., the function entry point of the `render_frame` function.

B.7 Binary Code Control Flow Obfuscations

This section covers the control flow obfuscations applied to the binary code by Diablo as described in Task T2.4 of the DoW. These obfuscations include, but are not limited to, control flow flattening, opaque predicate insertion, and the insertion of branch/call functions.

Rather than most protection techniques, binary code control flow obfuscations are not merely a binary enable/disable choice, but they can be applied a number of times on each code region. To allow the developer to have a more fine-grained control over the applications, we introduce additional production rules:

```
<PROTECTION_NAME>          ::=   binary_obfuscations

<OBFUSCATION_PARAMETERS> ::=   <ID> = <INTEGER> [ : <ID> = <INTEGER> ]*

<OBFUSCATION>              ::=   <ID> [ : <OBFUSCATION_PARAMETERS> ]

<LIST_OF_OBFUSCATIONS>     ::=   <OBFUSCATION> [ , <OBFUSCATION> ]*

<PROTECTION_PARAMETER>     ::=   enable_obfuscation( <LIST_OF_OBFUSCATIONS> )
                               | disable_obfuscations( <LIST_OF_OBFUSCATIONS> )
```

Semantics:

- The `<OBFUSCATION_PARAMETERS>` rule is used to specify an optional, ':'-separated list of nu-

meric parameters to an obfuscation technique.

- For the code fragments annotated with `enable_obfuscation`, the binary control flow obfuscation protection is instructed to try to apply the requested protections. If parts of the annotated code cannot be transformed with a requested obfuscation technique, the protection technique will not apply the requested obfuscation technique.
- If multiple obfuscation transformations are listed, they are applied in the order they are listed. If an earlier-listed technique would prohibit the application of a later obfuscation technique, the protection technique will not apply the later one.
- In the case of nested obfuscation annotations, the deepest-nested annotation takes precedence. Furthermore, obfuscations specified in the deepest nesting will be applied first.
- The `disable_obfuscations` annotation is an absolute prohibition of applying the listed techniques to the annotated region. Even if the ADSS would make a decision to apply more binary code control flow obfuscations, the obfuscation techniques listed as argument for `disable_obfuscations`-annotated regions will never be applied to those regions.
- While no final decision has yet been made on which binary control flow obfuscations will eventually be supported in ASPIRE, at least the following IDs for such techniques will be available: `flatten`, `opaque_predicate`, `branch_function`, and `call_function`.
- Parameters need not be specified: if left unspecified, the protection technique will choose an appropriate value for the parameter.
- All techniques can be parameterized with the parameter `percent_apply`, which expresses the (maximal) percentage of basic blocks in this code region on which the technique will be applied. This percentage is computed on the number of basic blocks in the original, untransformed code region.
- The `flatten` transformation has an additional possible parameter, `max_switch_size`. This expresses the maximum number of basic blocks that are connected to a single switch block in the code region.
- The annotations do not propagate across function calls.
- In addition to the above list of technique IDs, a meta-ID 'ALL' will be available to prohibit all obfuscations when used in combination with `disable_obfuscations`.
- The meta-ID 'ALL' can also be used in combination with the `enable_obfuscations` annotation. In that case, the order in which the obfuscation transformations are applied are determined by the protection technique. The 'ALL' meta-ID can be given a list of parameters. When the protection technique applies individual transformation techniques as part of applying this meta-ID, the transformations are passed the parameters supported by each of them.

Example: Suppose a developer would want to flatten a code region in a function that has a recognizable control flow structure, but does not want *any* binary obfuscations applied to a hot loop in the same function:

```
1  int function(int x) {
2    _Pragma("ASPIRE begin protection(obfuscations,enable_obfuscation("
3       "opaque_predicates:percent_apply=25,"
4       "flattening))");
5    if (x < 0) x = -x;
```

```
6     if (x & 1) x = x << 2;
7     _Pragma("ASPIRE end");

8     _Pragma("ASPIRE begin protection(obfuscations,"
9                                    "disable_obfuscations(ALL)))");
10    for (int i = 0; i < x; i++)
11      if(a)
12        x = do_something(x);
13    _Pragma("ASPIRE end");

14     _Pragma("ASPIRE begin protection(obfuscations,"
15       "enable_obfuscation(ALL:percent_apply=25:max_switch_size=3),"
16       "disable_obfuscation(branch_function))");
17    if (do_something(x) == x)
18      x = -x;
19    else
20      x++;
21    return x;
22    _Pragma("ASPIRE end");
23  }
```

In the above example, the control flow of lines 4 and 5 will be obfuscated as follows. First, 25% of the basic blocks of that code region will have an opaque predicate inserted. Secondly, this code region (with opaque predicates inserted) will be flattened (i.e., a switch block will be introduced, and all control flow is redirected through this block). Because we give no additional parameters for the flattening, all basic blocks in this code region will be redirected to the switch block (to the extent that this is possible).

The loop on lines 9-11 will not have any binary code obfuscation transformations applied to it, no matter what additional decisions

The annotations on lines 14-16 indicate that the code region on lines 17-22 can have all obfuscations applied to them, except branch functions. For each of the obfuscation transformations that is applied, 25% of the basic blocks are transformed. Furthermore, obfuscation transformations that have the `max_switch_size` property, such as flattening, will have this property set to the value of 3, the other techniques will not see this (for them unknown) parameter.

B.8 Client-Server Code Splitting by means of Barrier Slicing

The client-server code splitting protection is described in Section 3.3 of Deliverable D1.04. The corresponding annotations are the following ones:

```
<PROTECTION_NAME> ::= barrier_slicing

<PROTECTION_PARAMETER> ::= barrier(<LIST_OF_IDS>)
                         | criterion(<LIST_OF_IDS>)
                         | label(<ID>)
```

Semantics:

- Can specify either a set of barriers or a criterion:
  - `barrier(<LIST_OF_IDS>)` : a list of valid variable names that indicates the barriers for barrier slicing computation.
  - `criterion(<LIST_OF_IDS>)` : a list of valid variable names that indicates variables of the criterion for slicing.

- `label(<IDS>)` : indicates the set of barriers and criteria that belong to the same barrier slice computation.

```
1    int dd1;
2    int dd2;
3    int year1;
4    int year2;

5    void g() {
6      int y;
7      f(y);
      }

8    void f(int x) {
9      _Pragma("ASPIRE begin protection (barrier_slicing, barrier(year1),"
               "label(slicing1))")
10     year1 = read();
11     _Pragma("ASPIRE end")
12     int ref = year1;
13     int year2 = read();

14     for (int i = ref; i < year1; i++) {
15       if (i % 4 == 0)
16          dd1 += 1;
         }
17     dd1 = calculate_original();

18     dd2 = 0;
19     for (int i = ref; i < year2; i++) {
20       if (i % 4 == 0)
21          dd2 += 1;
         }
22     dd2 = calculate_current();

23     _Pragma("ASPIRE begin protection (barrier_slicing, criterion(dd1,dd2),"
               "label(slicing1))")
24     if (dd2 - dd1 > 30)
25       printf("Fail\n");
26     else
27       printf("Ok\n");
28     _Pragma("ASPIRE end")
     }
```

Example: The code computes two variables, dd1 and dd2, and compares the resulting values to enforce a license check. Annotation at line 9 denotes a new barrier on statement 10 for variable year1. Annotated block for the barrier ends at line 11. Attribute "label" is used to pair this annotation with corresponding criterion.

Annotation at line 23 indicates the beginning of a criterion, consisting of variables dd1 and dd2 at lines 24-27. Lines of code that are in the criterion are those that appear between the beginning of the annotation (line 23) and its end (line 28). The annotation also reports a label that corresponds to the correct barrier(s).

B.9 Code Mobility

The basic code mobility protection is explained in Section 3.4 of deliverable D1.04. Its corresponding source code annotations are the following:

```
<PROTECTION_NAME> ::= code_mobility

<PROTECTION_PARAMETER> ::=   status ( mobile | static )
                         |   data ( mobile | static )
```

Semantics:

- Case `status (mobile)`: specifies that the technique should be applied to the potential blocks in the code region.
- Case `status (static)`: specifies that the technique should not be applied to the code region.
- `data`: specifies whether the read-only data associated with the code is to be made mobile as well or not (analogous to the 'status' parameter).

Example: This example shows how to instruct the framework to protect a code region using code mobility:

```
int f(int x) {
  _Pragma("ASPIRE begin protection(code_mobility,status(mobile))");

  if(license)
     execute_sensitive_code();

  return x;
  _Pragma("ASPIRE end");
}
```

B.10 Remote Attestation

The use of remote attestation (RA) to protect software is described in Section 4.8.3 of deliverable D1.04. The corresponding annotations are as follows:

```
<PROTECTION_NAME> ::= remote_attestation

<PROTECTION_PARAMETER> ::=   static_ra(<LIST_OF_IDS>)
                         |   static_ra_region
                         |   dynamic_ra_variable(<ID>)
                         |   dynamic_ra_invariant(<PEXPR>)
                         |   dynamic_ra_autodiscovery
                         |   implicit_ra(<LIST_OF_IDS>)
```

These annotations permit the specification of either a generic or a concrete remote attestation technique.

`static_ra(<LIST_OF_IDS>)`: concrete type of RA. It defines a new attestator to which it is possible to assign code regions to be monitored. This annotation parameter accepts a list of IDs according to the following pattern:

```
<LIST_OF_IDS> ::=  RW_{NORMAL|GOLDBACH},
                   HF_{BLAKE|MD5|SHA1|SHA256|RIPEMD160},
                   NI_{1|2|3|4},
                   NG_1,
                   MA_1,
                   DS_1
               |   <INTEGER>
```

It is possible to use either the extended notation which explicitly defines each static remote attestation fundamental block or the concise notation which specifies the attestator version as an integer. Each version is associated to a permutation of the extended notation, hence it accepts values from 1 to 40 (as described in Section 1.4.1 of deliverable D3.06). When the `static_ra` parameter is used it is mandatory to specify a set of additional protection parameters as follows:

```
<PROTECTION_PARAMETER> ::=  label(<ID>)
                        |   frequency(<VALUE>)
```

The `label` parameter specifies the attestator label. This label is used to refer to the attestator whenever a code region has to be assigned to it.

The `frequency` parameter specifies the attestation frequency for the defined attestator. It means that the server sends one attestation every `<VALUE>` seconds to this attestator.

`static_ra_region`: concrete type of RA. It defines a new code region that will be monitored by static remote attestation. When this annotation is used it is mandatory to specify also the following protection parameters:

```
<PROTECTION_PARAMETER> ::=  attestator(<ID>)
                        |   attest_at_startup(true|false)
```

The `attestator` parameter accepts a `<ID>` that specifies the label of the attestator to which this code region will be associated. Note that it is not possible to define any code region without having defined an attestator to which assign the region.

The `attest_at_startup` parameter specifies whether the code region will be attested as soon the application is launched or not.

Example: this example shows how a new attestator is defined (line 13) and how to define a new code region to be attested (line 4 and 8). The attestation code region annotation encloses the code that must be monitored by means of static remote attestation and assigns the region to the attester labeled as `first_attestator`.

```
1  int f(int max) {
2      int i,sum = 0;
3      int y ;

4      _Pragma("ASPIRE begin protection(remote_attestation,
              static_ra_region,
              attestator(first_attestator), attest_at_startup(true))")
5      for(i=0; i<max; i++){
6          y=2*max;
       sum+=y;
7      }
8      _Pragma("ASPIRE end");
```

```
 9      return sum;
10  }

11  int main()
12  {
13      _Pragma("ASPIRE begin protection(remote_attestation,
              static_ra(RW_NORMAL, HF_BLAKE2 , NI_1, NG_1, MA_1, DS_1),
              label(first_attestator), frequency(10))")
14      _Pragma("ASPIRE end");

15      x = 33;
16      printf("Sum=%d",f(x));
17      return 0;
18  }
```

dynamic_ra_variable: concrete type of Dynamic RA. It allows to define and label a variable in order to make it usable to define invariants inside the application. The `<ID>` passed to this annotation parameter must uniquely identify the variable over the entire application.

dynamic_ra_invariant: concrete type of Dynamic RA. It allows to specify an invariant as a logic expression that involves constants values and variables' labels defined by using the `dynamic_ra_variable` annotation. The provided invariant will be translated and used to attest the integrity of the application.

Example: this example shows how a developer can ask the framework to protect a code region by means of invariants monitoring RA. Only one invariant is defined (line 4 and 9), which states that the sum of the values of variables `x` and `y` must be less than 100 for the enclosed portion of the program (i.e., line 5 to 8). Note that, variables IDs used in the invariant definition have been tagged (line 3 and line 14).

```
 1  int f(int max) {

 2      int i,sum = 0;
 3      int y __((ASPIRE("protection(remote_attestation,
                                 dynamic_ra_variable(y))")));

 4      _Pragma("ASPIRE begin protection(remote_attestation,
                                     dynamic_ra_invariant(x+y<100))");

 5      for(i=0; i<max; i++){
 6          y=2*max;
 7          sum+=y;
 8      }

 9      _Pragma("ASPIRE end");

10      return sum;
11  }

12  int main()
13  {
14      int x __((ASPIRE("protection(remote_attestation,
                                 dynamic_ra_variable(x))")));
15      x = 33;
16      printf("Sum=%d",f(x));
```

```
17      return 0;
18  }
```

dynamic_ra_autodiscovery: concrete type of Dynamic RA. It allows to specify a code region as protected by invariants monitoring. A code region that is enclosed by this annotation will be analyzed in order to automatically discover all the possible invariants. Then the invariants and the needed variables are identified and used at runtime to attest the code region.

Example: the code sample reported hereafter shows how a code region can be annotated in order to require that invariants monitoring will automatically protect it (line 4 and 9). Note that the annotation only specifies the region which has to be protected (line 5 to 8), the discovery of invariants and the variables identification will be transparently performed.

```
 1  int f(int max) {
 2      int i,sum = 0;
 3      int y;

 4      _Pragma("ASPIRE begin protection(remote_attestation,
                                      dynamic_ra_autodiscovery");

 5      for(i=0; i<max; i++){
 6          y=2*max;
 7          sum+=y;
 8      }

 9      _Pragma("ASPIRE end");

10      return sum;
11  }

12  int main()
13  {
14      int x;
15      x = 33;
16      printf("Sum=%d",f(x));
17      return 0;
18  }
```

B.11 Control Flow Tagging

The tamper detection technique of CFT has been presented in detail in Section 4.3 of deliverable D1.04.

The grammar specified below allows to specify where attestators and verifiers of the CFT protection technique should be placed in the code. Attestators are the place where the gates' counters have to be incremented. Verifiers are the places in the code where a rule is checked. A rule is a boolean expression that combines the gates' counters.

```
    <PROTECTION_NAME> ::= cf_tagging

    <PROTECTION_PARAMETER> ::= gate(<ID>)
                             | <VERIFIER>
```

```
<VERIFIER> ::= check (<BOOL_EXPRESSION>)
             | location ( <LOCATION> )

<BOOL_EXPRESSION> ::= <SQ_STRING>

<LOCATION>  ::= local | remote

<REACTION>  ::= exit | 1..8
```

The user has to specify markers in the code. These markers indicate the location of the gates to be installed. The user also has to indicate the location of verifier(s). In addition to verifiers' locations, their verification rules have to be provided, indicating by which gates the execution flow is supposed to have passed. A rule is a boolean expression that is passed as a string.

Semantics:

- `ID`: specifies the gate's ID, in order to be able to use it later in the verifier's boolean expression.
- `BOOL_EXPRESSION`: this expression forms as a C-language legal boolean expression. It specifies which gates to be activated and be verified. A gate ID refers to the value of the counter of the gate. It leads to false in this expression if it is equal to 0, true otherwise. Counter values can be compared with comparison operators and immediate integer values.
- `LOCATION`: the location of the checking must be specified on only on verifier makers, not on gate marker.
- `REACTION`: 'exit' to abruptly quit from the program, or put '1..8' as time bombs reaction level.

Example: The following example shows how a conditional path can be specified. Note the use of a boolean expression specified in the verifier:

```
// ...
// subsequent execution passes through gate1 or gate 2, never both
if (bool_val)
{
  _Pragma("ASPIRE begin protection(cf_tagging,gate(main_if))");
  //...
  _Pragma("ASPIRE end");
}
else
{
  _Pragma("ASPIRE begin protection(cf_tagging,gate(main_else))");
  //...
  _Pragma("ASPIRE end");
}

function1();

// ...

// we want to check here that execution went through
// gates main_if OR main_else, AND gate function1
// Verifier code is to be generated in the application, not on server

_Pragma("ASPIRE begin protection(cf_tagging,"
         "check('(function1 > 0 && (main_if != main_else))'),
```

```
            location(local),reaction(8))");

  //...
  _Pragma("ASPIRE end");

void function1()
{
  // ...
  _Pragma("ASPIRE begin protection(cf_tagging,gate(function1))");
  // ...
  _Pragma("ASPIRE end");
}
```

B.12 Software Time Bombs

Time bombs is a protection that has a delayed damaging action on the execution code, as described in Section 4.7 of deliverable D1.04. The grammar enables to notify where in the code the protection method is authorized to insert *incrementors*.

Initialization annotation is necessary to pre-process all annotated data variable. This annotation shall put in native code that always be executed, and as early as possible before other time bombs annotation.

```
     <PROTECTION_NAME> ::= timebombs

     <PROTECTION_PARAMETER> ::= init
```

The area of code specified should have a high probability to be activated. The protection will analyze the graph to set the incrementors at the most relevant locations based on these begin/end notations specified by the user.

```
     <PROTECTION_NAME> ::= timebombs

     <PROTECTION_PARAMETER> ::= code_area(<ID>)
```

Semantics:

- `ID`: This identifier can be used by the application developer to specify on the time bombs data attribute as specified in Section A.6 what is his favorite code area to insert incrementors.

Example: We refer to the example code in Section A.6.

B.13 Anti-cloning

The anti-cloning technique as presented in Section 4.5 of deliverable D1.04, comprises a sync with the ASPIRE security server, which is independent of any other technique or event. The sync event can be invoked at any point in time during the execution of the ASPIRE protected application and from any place in the application. When invoked the anti-cloning mechanism does not need any parameters; only the hooks from where it will be invoked need to be defined. The user has to specify these in the code.

```
<PROTECTION_NAME> ::= anti_cloning
<PROTECTION_PARAMETER> ::=   status
                           | decision(<SQ_STRING>)
```

Semantics:

- SQ_STRING: This string is the name of the variable holding the result of the anti-cloning check done on server side. The variable must be of int type.

Examples The following example shows how the code must be written and transformed in order to send an anti-cloning status.

```
  __attribute__(ASPIRE("protection(anti_cloning, status)"));
```

Code is transformed as follows:

```
/* at the beginning of the file */
void antiCloningSendStatus(void);

/* replacing the annotation */
  antiCloningSendStatus();
```

The following example shows how the code must be written and transformed in order to get an anti-cloning decision. The function antiCloningGetDecision() returns 0 if the server considers the client as valid, and non-0 if it considers it as a clone.

```
  int   response = 0;
  __attribute__(ASPIRE("protection(anti_cloning, decision(response))"));

  if (response)
  {
    /* display a pop-up "You are a clone" */
  } /* if */
```

Code must be transformed as follows:

```
/* at the beginning of the file */
int antiCloningGetDecision(void);

  int   result = 0;
/* replacing the annotation */
  result = antiCloningGetDecision();

  if (result)
  {
    /* display a pop-up "You are a clone" */
  } /* if */
```

C JSON Format for Diablo - X-translator Interface

Section authors:
Bjorn De Sutter, Sander Bogaert, Jonas Maebe, Jens Van den Broeck (UGent), Andreas Weber (SFNT)

Diablo produces a description of the native code chunks in the form of JSON files (BLC02). The specification for this interface defines several data types:

```
instruction {
   "type" : string ("normal", "address_producer", "constant_producer")
   "regswritten" : array of register names
   "regsread" : array of register names

   // instruction-specific fields (type "normal")
   "encoding" : string (hexadecimal representation of encoded instruction)

   // address producer-specific fields (type "address_producer")
   "addrregister" : register name
   "addrsymbol" : integer (index in the symbols array)

   // constant producer-specific fields (type "constant_producer")
   "targetregister" : register name
   "valuesymbol": integer (index in the symbols array)
}

basic_block {
   "instructions" : array of instruction objects
   "regsliveout" : array of register names
   "function name" : string (name of the function this basic block belongs to)
   "function offset" : integer (offset of this basic block relative to the start of the function)
}

edge {
   "type" : string ("fallthrough", "call", "return", "switch", "jump", "jump-eq", ...)
   "sourcebbl" : integer (position of bbl in "bbls" array)

   // in case the edge is internal
   "destbbl" : integer (index in the "bbls" array)

   // in case the edge is external
   "destsymbol" : integer (index in the symbols array)
}

chunk {
   "bbls" : array of basic_block objects
   "edges" : array of edge objects
   "regslivein" : array of register names

   // in case this chunk is to be made mobile
   "mobile_id" : integer (chunk UID)
}

symbol {
   "name" : string

   // depending on the referrer:
   //  - address producer: address of the target symbol;
   //  - constant producer: value of the produced constant;
   //  - edge: address of the target basic block.
   "address" : string (hexadecimal value)
}

top_level (unnamed) {
   "chunks" : array of chunk objects
   "symbols" : array of symbol objects
}
```

A small example here shows how the above data types are used to export chunks. The example is the result of the current, work-in-progress, Diablo extraction process and a lot of fields are not

used or filled yet:


```
{
  "chunks":[
    {
      "bbls":[
        {
          "instructions":[
            {
              "type":"normal",
              "encoding":"e28db004",
              "regswritten": ["R11"],
              "regsread": ["R13"],
              "addrsymbol":"unimplemented"
            },
            {
              "type":"normal",
              "encoding":"e24dd008",
              "regswritten":["R13"],
              "regsread":["R13"],
              "addrsymbol":"unimplemented"
            },
            {
              "type":"normal",
              "encoding":"e50b0008",
              "regswritten":[],
              "regsread":[R0","R11"],
              "addrsymbol":"unimplemented"
            },
          ],
          "regsliveout":["R4", "R5", "R6", "R7", "R8", "R9", "R10", "R11", "R13", "R15", "CPSR",
                         "SPSR", "f0", "f1", "f2", "f3", "f4", "f5", "f6", "f7", "fpsr", "s0",
                         "s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s10", "s11",
                         "s12", "s13", "s14", "s15", "s16", "s17", "s18", "s19", "s20", "s21",
                         "s22", "s23", "s24", "s25", "s26", "s27", "s28", "s29", "s30", "s31",
                         "fpscr", "fpsid", "fpexc", "d16", "d17", "d18", "d19", "d20", "d21",
                         "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30", "d31"]
        }
      ],
      "regslivein":["R0", "R1", "R4", "R5", "R6", "R7", "R8", "R9", "R10", "R13", "R15", "CPSR",
      "SPSR", "f0", "f1", "f2", "f3", "f4", "f5", "f6", "f7", "fpsr", "s0", "s1", "s2", "s3",
      "s4", "s5", "s6", "s7", "s8", "s9", "s10", "s11", "s12", "s13", "s14", "s15", "s16", "s17",
      "s18", "s19", "s20", "s21", "s22", "s23", "s24", "s25", "s26", "s27", "s28", "s29", "s30",
      "s31", "fpscr", "fpsid", "fpexc", "d16", "d17", "d18", "d19", "d20", "d21", "d22", "d23",
      "D24", "D26", "D28", "d30", "d31"]
    }
  ]
}
```

List of abbreviations

ACCL ASPIRE Client-side Communication Logic

ACTC ASPIRE Compiler Tool Chain

ADS Attestation Data Structures

ADSS ASPIRE Decision Support System

AES Advanced Encryption Standard

AKB ASPIRE Knowledge Base

API Application Programming Interface

ARM ADS ARM Developer Suite

ARM RCVT ARM Real View Compilation Tools

ARM RVDS ARM RealView Development Suite

ARM NEON NEON is a trademark from ARM, not an acronym

ASPIRE Advanced Software Protection: Integration, Research, and Exploitation

BCxx Binary code document nr. x

BLPxx Binary-level software processing step nr. xx

BLCxx Binary-level configuration file nr. xx

BLLxx Binary-level log file nr. xx

CBC Cipher Block Chaining

CDT C Development Toolkit

CFG Control Flow Graph

CFT Control Flow Tagging

CPU Central Processing Unit

DCL Diversified Crypto Library

DES Data Encryption Standard

DES3 Triple DES

DOP Number of destination register operands

DoW Description of Work

DPL Dynamic Program Length

DSL Domain Specific Language

DST Number of destination operations

Dxx Datum produced or used by the ASPIRE ACTC identified wit the nr.xx

Dx.y ASPIRE deliverable # y in workpackage x, y is a two digit number

EBNF Extended Backus–Naur Form

ECB Electronic Code Book

EDG Number of edges

ELF Executable and Linkable Format

EMF Eclipse Modeling Framework

GCC GNU C Compiler

GNU GNU is Not Unix

ID Identifier

IP Internet Protocol

IRI Internationalized Resource Identifier

JSON JavaScript Object Notation

KB Knowledge Base

L1P Level 1 Protections

L2P Level 2 Protections

LLVM Low-Level Virtual Machine (now has lost that meaning)

MATE Man-at-the-end

MILP Mixed Integer Linear Programming

MITM Man-in-the-middle

Mx Month x. A reference to a specific time in the ASPIRE project. It refers to the x'th month since November 2013.

OS Operating System

OWL Web Ontology Language

PI Protection Instantiation

PN Petri Net

PO Protection Objective

RA Remote Attestation

RAP Rich Ajax Platform

RCP Rich Client Platform

RNC Residue Number Coding

SDK Software Development Kit

SHA1 Secure Hash Algorithm 1

SIMD Single Instruction Multiple Data

SLPxx Source-level software processing step nr. xx

SLCxx Source-level configuration file nr. xx

SLLxx Source-level log file nr. xx

SP Self-Profiling

SSH Secure Shell

SVN Subversion

TCP Transmission Control Protocol

TXL This is not an acronym (see `http://www.txl.ca/nwhy.html`)

UI User Interface

VFP Vector Floating Point

VM Virtual Machine

WB White-Box

WBT White Box Tool

WBTA White Box Tool for ASPIRE

WPx Work Package x

XML eXtensible Markup Language

X-translator Cross-translator

References

[1] Cataldo Basile, Canavese Daniele, Jerome d'Annoville, Bjorn de Sutter, and Fulvio Valenza. Automatic discovery of software attacks via backward reasoning. In *SPRO 2015: 1st International Workshop on Software Protection*, pages 52–58. IEEE Computer Society, 2015.

[2] Murray Campbell, A. Joseph Hoane Jr., and Feng-hsiung.Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.

[3] ASPIRE Consortium. D1.02 - aspire attack model.

[4] Free Software Foundation. Gnu compilers documentation, section 6.3 - attribute syntax, 1988-2014.

[5] Free Software Foundation. Gnu compilers documentation, section 7 - pragmas, 1988-2014.

[6] Donald E. Knuth. Generating all n-tuples. In *The Art of Computer Programming – Volume IV*. Addison-Wesley, 2004.

[7] Donald E. Knuth. Generating all permutations. In *The Art of Computer Programming – Volume IV*. Addison-Wesley, 2004.

[8] Donald E. Knuth. Generating all combinations. In *The Art of Computer Programming – Volume IV*. Addison-Wesley, 2005.

[9] Leonardo Regano, Canavese Daniele, Cataldo Basile, Alessio Viticchie', and Antonio Lioy. Towards automatic risk analysis and mitigation of software applications. In *WISTP 2016: 10th International Conference on Information Security Theory and Practice*, pages 120–135. Springer International Publishing, 2016.

[10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.