Advanced Software Protection:
Integration, Research and Exploitation

D5.01
Framework Architecture, Tool Flow, and APIs
of the ASPIRE Compiler Tool Chain and Decision Support System

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 / D5.01
WP and tasks contributing:	WP 5 / Task 5.1
Due date:	July 2014 – M09
Actual submission date:	17 September 2014

Responsible Organization:	POLITO
Editor:	Cataldo Basile
Dissemination level:	Confidential
Revision:	1.0

Abstract:
This deliverable is the first report on the design and development of the ASPIRE Compiler Tool Chain (ACTC) and the ASPIRE Decision Support System (ADSS). It first introduces the ASPIRE source code annotations. Then it presents the current ACTC status, on both the source level and the binary rewriting parts, and the ongoing long-term design effort. Finally, it focuses on the ADSS architecture, workflow, and shows an initial ADSS specification.

Keywords:
ACTC; ADSS; tool flow; decision support; ASPIRE annotations.

Editor
Cataldo Basile (POLITO)

Contributors (ordered according to beneficiary numbers)
Bjorn De Sutter (UGent)
Bart Coppens (UGent)
Sander Bogaert (UGent)
Jonas Maebe (UGent)
Daniele Canavese (POLITO)
Rachid Ouchary (POLITO)
Brecht Wyseur (NAGRA)
Mariano Ceccato (FBK)
Roberto Tiella (FBK)
Andrea Avancini (FBK)
Alessandro Cabutto (UEL)
Werner Dondl (SFNT)
Andreas Weber (SFNT)
Jerome D'Annoville (GTO)

The ASPIRE Consortium consists of:

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

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

Disclaimer The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 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 deliverable is the first report on the design and development of the ASPIRE Compiler Tool Chain (ACTC), which is developed in Task T5.1, and the ASPIRE Decision Support System (ADSS), which is developed in Task T5.2.

There are three major contributions in this deliverable: (1) the specification of the initial ASPIRE annotations, (2) the presentation of the ACTC for D5.02 and D5.03 at M12, as currently being integrated, and of the ongoing long-term ACTC design effort, and (3) the ADSS specifications.

Relying on attributes and pragmas, which are code annotation mechanisms supported by most compilers, the ASPIRE annotations are specified semi-formally. Their general semantics are specified as well. The annotations include concrete annotations of protections to be applied on data and code, security requirement annotations with which users can annotate the assets in their software and describe the threats against which protections need to be applied, and so-called profile annotations that will make the use of annotations less cumbersome. For many of the concrete ASPIRE protections developed in the project, concrete annotations are formalized and illustrated on examples. Furthermore, a file format specification details how annotation information extracted from the source code files will be passed to the different tools in the ASPIRE tool chain.

The ACTC consist of three major parts: the source-level tools, a standard compiler, and the binary-level tools. To visualize their components and flow uniformally, a template has been designed.

The source-level tool flow design for M12 consists of ordered passes for (1) code preprocessing and code normalization, (2) white-box cryptography, (3) annotation extraction, and (4) data hiding obfuscations.

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 of the ACTC for M12 operates in four steps: (1) code analysis and extraction, (2) additional code generation, (3) integration of that code, (4) and binary-level obfuscation of all integrated code, with a clear separation of concerns between different tools in the ACTC that implement these steps.

Design revisions for later versions of the M12 ACTC have already been designed, and are in line with the overall flow and design options taken for the M12 ACTC.

A virtual machine environment that is shared with all partners. This environment contains all the necessary tools to enable all partners to reproduce result easily, as well as to collaborate on the integration of the individual protection techniques of the different partners.

The ADSS is the ASPIRE-developer software component whose aim is to support software developers when selecting the most suitable set of protection techniques that protect the application assets and that satisfy a set of developers' constraints. The ADSS is a complex and modular software system that takes as input the source code of the application to protect with the provided annotations, and generates a set of instructions for the ACTC to deploy the most suitable protection, named golden combination. The ADSS also outputs logs and reports that explain the developers the entire decision process.

The golden combination is selected with a complex workflow. Initially, all the information about the application to protect is gathered using compiler tools. Then, to identify the suitable set of protections, the gathered information is enriched by means of a set of logical inferences. Examples of these inferences are the automatic identification of the attack paths against the application assets, and the deduction of protections' usability to mitigate risks against the assets. The logical inferences will be implemented as Enrichment Modules within the ASPIRE Enrichment Framework. The identified suitable combinations are used to generate an optimization program, whose solution, computed with off the shelf solvers, is the golden combination that will be processed to derive the instructions for the ACTC.

Contents

List of Figures

List of Tables

1 Introduction

Section authors:
Bjorn De Sutter, Cataldo Basile

1.1 Role of this Deliverable

The main goals of Work Package 5 in the ASPIRE DoW are the design and development of the ASPIRE Compiler Tool Chain (ACTC) in Task T5.1 and the ASPIRE Decision Support System (ADSS) in Task T5.2, including their interaction to automate as much as possible the protection of assets in software distributed as native binary programs or as native libraries. Deliverable D5.01 is the first report on the design of the ADSS and ACTC.

From a high-level perspective, the eventual ACTC and ADSS will operate as depicted in Figure 1. Apart from configuration files as needed by all software development tools (not drawn in the figure), their input consists of source code written in C or C++ annotated with so-called *security requirement annotations*. These annotate the assets (code and data) to be protected, and they identify the threats that need to be mitigated, as described in Section 4.

Compiler (analysis) tools in the ACTC will then extract the annotations and compute a wide range of relevant data about the software regions to be protected. All extracted and computed information is then made available to the ADSS to allow it to determine the so-called *golden combination* of protections to be applied. On the basis of this selection, the ADSS will update the annotations in the source code itself (or their representation in the annotation fact database), turning them into so-called *protection annotations* and tool configuration files, on the basis of which compiler (transformation) tools in the ACTC will then transform the software. The output will consist of reports from the ADSS that document the decisions made by the ADSS regarding the golden combination, reports from the ACTC about the transformations that were actually applied, a protected binary program or library, and components to be deployed on the ASPIRE Security Server (see D1.04).

The architecture and internal operation of the protected binary and the server-side components is documented in ASPIRE Deliverable D1.04. The role of this deliverable is to document the design of the source-code annotations (Part I), the design of the initial ACTC for M12 as well as the ongoing longer-term design effort (Part II), and the ongoing design of the ADSS (Part III).

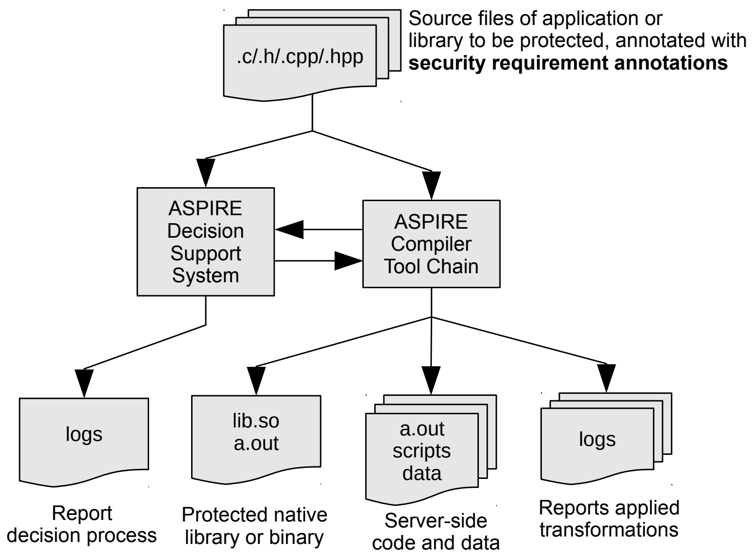
.c/.h/.cpp/.hpp
Source files of application or library to be protected, annotated with **security requirement annotations**
ASPIRE Decision Support System
ASPIRE Compiler Tool Chain
logs
Report decision process
lib.so a.out
Protected native library or binary
a.out scripts data
Server-side code and data
logs
Reports applied transformations

Figure 1: High-level perspective on the eventual ASPIRE outcomes of WP5.

1.2 Overall Planning and Progress of Work Package 5

The deliverables and milestones related to WP5 as described in the DoW describe two (mostly) concurrent design tracks.

The first track in (Task T5.1) is the development of the ACTC from a simple, basic tool flow that can handle toy code examples without really protecting them, over a version that applies offline protections to the project uses cases of WP6, to versions that also apply the online protection techniques and later also the renewability protection techniques to the full use cases.

The second track (Task T5.2) concerns the development of the ADSS and the integration of the ADSS and the ACTC.

In practice, our efforts in WP5 in the first year have been focused on three aspects:

Design and Implementation of the ADSS In the first year, the focus is on modeling all the required knowledge in the knowledge base on which the ADSS will build. The status is documented in Part III of this document. It builds on the ASPIRE security modeling documented in D4.01.

Continuous Integration In line with the first track mentioned above, we are implementing a simple tool flow version, that will gradually be extended into the full-blown ADSS and ACTC.

Long-term Design While working on the simple, initial versions of the tool flow, we are already discussing and drafting the eventual design of the full-blown ADSS and ACTC, such that we know where the continuous integration is heading, and such that the complete versions will be implemented in time for demonstration in WP6 at the end of the project.

The progress in the continuous integration effort and in the long-term design efforts is driven by two different activities.

On the one hand, there is the ongoing development of tool chain components that implement individual protections or combinations of protections and that gradually become ready for integration in the tool flow. This is mostly the case for offline protections developed in WP2, since WP2 started at the beginning of the project. We are happy to be able to state that **at this point in time, more components are ready for initial integration than foreseen in the DoW. The ACTC that will be reported at the end of year 1 of the project will therefore already support more protections than anticipated in the DoW.** Also, it will already integrate all the major compiler tools from the four major compiler tool developers in the consortium:

- the source code sanity checker (FBK);
- the source code analysis tool CodeSurfer;
- the annotation extraction tool (FBK);
- the white-box encryption code generation tool (NAGRA);
- the source-to-source rewriter (FBK);
- the standard compiler and linker;
- the Diablo link-time rewriter (UGent);
- the binary-to-bytecode cross-translator (SFNT);
- the SoftVM (SFNT).

Compared to Figure 1, the M12 compiler version will look like the one depicted in Figure 2. The major limitations are that only offline protections are supported, and that there is no decision support yet. The latter implies that the annotations in the source code have to describe the concrete protections to apply, instead of the assets to protect. For that reason, these annotations are named *protection annotations* in the figure. Those annotations are described in Section 2.3.

On the other hand, there are the growing insights into all the requirements and relevant issues concerning the full-blown tool chain. These insights grow as a result of the ongoing exchanges of ideas and discussions based on insights obtained through individual and collaborative experimentation and reflection. The maturing of ideas within the consortium is particularly important with respect to the more challenging online protections, of which the RTD started only in M6, and with respect to the interaction between the ADSS and the ACTC. In this regard, we have come to the conclusion that at this point in time, it remains a rather open challenge to concretely design that interaction, basically because more research, experimentation, and discussion are needed to decide on the exact split of responsibilities between individual components in the ACTC on the one hand, and the ADSS on the other hand.

To make this issue more concrete, let's consider the example of a large, security-sensitive procedure `f()` in some program. Ideally, the user of the ADSS and ACTC should only have to annotate this procedure as security-sensitive, and specify its protection and possible performance constraints. On that basis, the ADSS and ACTC should be able to select the best combination of protections for this procedure, and then configure the compiler tools to apply the selected protections. The question to answer is the level of detail at which the ADSS should configure the tools to apply protections. Two extremes examples of such configurations are the following:

- The ADSS specifies that the ACTC should replace 10% of the native code of `f()` by bytecode to be interpreted by an embedded VM (protection of Task T2.3), by inserting code guards that executes the checks "very frequently" on "large" code fragments, and that binary code obfuscations should be applied "lightly". In that case, the ACTC has to decide itself which 10% of the code to replace by bytecode, and where to insert guards and binary code obfuscation in `f()`. All of these decisions will be based on code analysis and profile information in the ACTC components.

  In this approach, one can easily understand that the ADSS only needs to search a limited search space, consisting, e.g., of the choices between 0%, 10%, 20%, ..., and 100% translation to bytecode; between "no", "rare", "frequent" and "very frequent" guards on "small", "medium" or "large" code fragments; and between "light", "medium", and "heavy" binary code obfuscation. The risk is of course that suboptimal results will be obtained.

  Furthermore, one can envision that in order to reach a (close to optimal) decision, the ADSS does not need to know all details of the procedure code to protect and all detailed code analysis results. Instead, the compiler (analysis) tools in the ACTC could compute or estimate

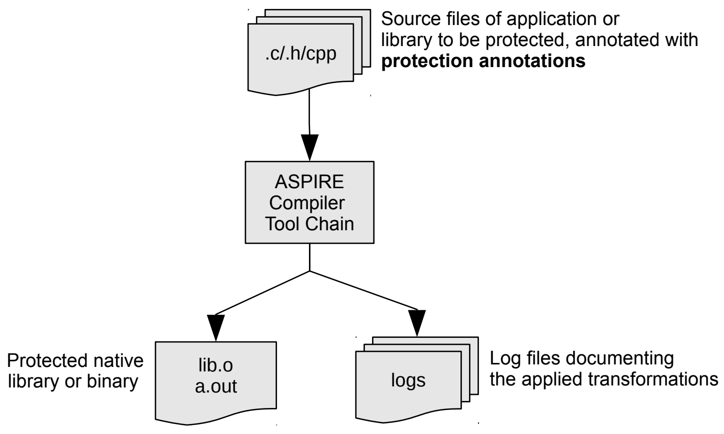
.c/.h/cpp
Source files of application or library to be protected, annotated with **protection annotations**
ASPIRE Compiler Tool Chain
Protected native library or binary
lib.o
a.out
logs
Log files documenting the applied transformations

Figure 2: High-level perspective on the first tool chain version developed in WP5.

the impact of applying the different levels of protections on the code at hand, summarize that information for the ADSS in terms of the relevant metrics (as designed in Task T4.2 of the project), and let the ADSS make a decision based on that summary information. The risk is then again that suboptimal results will be obtained, in particular when the estimates passed to the ADSS by the compiler tools would prove not be composable.

- Alternatively, the ADSS specifies that the ACTC should replace instruction sequences $i_{11} - i_{13}$ and $i_{24} - i_{42}$ by bytecode, that at points x and y in the procedure, guards have to be inserted that guard the code fragments $i_6 - i_{19}$ and $i_{17} - i_{33}$, and that an obfuscating branch function has to be inserted at point w, while the control flow flattening obfuscation has to be applied on blocks a, b, and c in the procedure's control flow graph.

  In this approach, the ADSS has a huge search space, because it needs to decide on all little details. Furthermore, it needs to be capable of collecting all the little details of all protection implementations (such as pre- and postconditions for applying transformations in a valid manner) and of reasoning about them.

Clearly the first extreme is likely to fail, because decisions about individual protections, and estimates of their impact on the relevant metrics, will likely turn out not to be composable in many cases.

For example, when estimating the impact of translating 10% of the native code of `f()` to bytecode, the ACTC might have assumed that it would translate the 10% least frequently executed code fragments in `f()`. To estimate the impact of "lightly" applied control flow obfuscations, the ACTC might also have considered applying those obfuscations to the least frequently executed code. The problem is that if this information is not passed to the ADSS, the ADSS has no way to know how well the two configurations compose, because the ADSS does not know which parts of `f()` are actually involved in each of the protections.

But also the second approach is sure to fail, if alone because the consortium cannot engineer such a massive ADSS within the resources of the project.

To solve this problem, we envision an approach close to what is described in the first example above, but in which more information than in the first example will be passed to the ADSS by the ACTC, and more fine-grained configurations will be selected. The extra, more detailed information passed from the ACTC to the ADSS can come in the form of more extended summary information (i.e., containing additional forms of information), or it can come in the form of summary information for much smaller code regions (such that information about a whole region alone suffices for the ADSS to decide whether or not protections in it are composable), or in combinations of both.

Finding the right balance is the subject of future research in WP5. With the current state of understanding within the consortium of all relevant aspects, it is not yet possible to make a definitive choice and to define the exact boundary between between ADSS and ACTC responsibilities. **It is hence not yet possible to document APIs and information flows as envisioned for the full-blown ASPIRE ADSS and ACTC.**

1.3 Structure of this Deliverable

In Part I, Section 2 will first present the overall design, the mechanisms, and the basic concepts on which the ASPIRE source code annotations will be built. The following three sections will then document the currently designed specifications of source code annotations for data-specific protections (Section A), code-specific protections (Section B), and security requirements (Section 4). Finally, Section 5 documents the specification of the annotation fact storage format, i.e., how the information encoded in the annotations will be represented once they are extracted from the source code and passed to the tools in the ACTC and ADSS.

In Part II, the short-term ACTC design is presented. Section 6 first presents the template for visualizing the tool chain components in ASPIRE. Section 7 will then discuss the source-level part of this tool flow that is currently being implemented and that will be the subject of Deliverables D5.02-03 (M12) and D5.04 (M18), after which Section 8 discusses the compilers, assemblers and linkers used in the project, and Section 9 discusses the binary rewriting part of the ACTC in more detail. Next Section 10 presents a sanity checking tool we developed that allows ACTC users to assess whether their source code meets the necessary requirements imposed by the different tools in the ACTC. Section 11 presents the virtual machine approach that we have developed to allow the creation of an identical build environment at all consortium partner sites, which will facilitate the continuous integration of new techniques into the ACTC and ADSS. Finally, Section 12 discusses the current status of the ongoing long-term design effort in WP5.

In Part III, the ADSS architecture and workflow is presented together with the ADSS components and their initial APIs. Section 13 introduces the problem the ADSS has to face, the ADSS functionalities, the ADSS workflow, the phases it has to perform and the research issues that need to be addressed. Finally, it presents the overall ADSS architecture and plans for the future implementation during the ASPIRE project. One of the main tasks the ADSS has to perform is a sophisticated reasoning about the application to protect which has the objective to determine the combinations of protections that can mitigate the attacks against the application. Therefore, Section 14 is devoted to the presentation of a set of examples of deductions and reasonings that can be performed to automatically determine how to protect an application. These reasonings are fragments of the inferential systems we aim at building to determine how to protect the target application. Sections F and G present support material that formally defines the examples in Section 14. This inferential system will be implemented by means of an Enrichment Framework, which is presented in Section 15. Section E presents some technical background on Description Logics, which are used for our inference system, and on linear programming, which is used to define the ASPIRE decision problem. Finally, Section H presents the initial specification of all the components that form the ADSS, including the initial API. This specification is important for the ADSS development. It will be constantly maintained, as it is the main way to permit third parties contributions.

1.4 Terminology

This document describes tools that will be "used" to protect applications that are written by what are typically called "developers". When referring to "users" in this document, and unless indicated otherwise explicitly, we refer to users of the tools, which will typically (but don't have to) be the same people as the developers of the original software to be protected. So the terms "user" and "developer" can mostly be interchanged and point to the same people.

Furthermore, in this document we will often refer to the "application" to be protected. At any point, this "application" can be either a binary executable (i.e., the main component of an application) or a dynamically linked library.

Part I

Source-code Annotations

2 Annotation Basics

Section authors:
Roberto Tiella

At the end of the project, ASPIRE aims for providing an ACTC and an ADSS that can be used by non-security experts to protect their applications. Those users will first 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 will automatically determine a suitable configuration to invoke the ACTC components on the code to be protected. For this purpose, we need to design a set of annotations that allows the ASPIRE user to annotate his assets and threats.

In practice, that ultimate goal of fully automated, non-assisted protection will at best be reached at the end of the project, and in fact it is an open question to what extent we will in practice be able to completely abandon all user assistance. Quite likely, some (expert) user assistance or guidance might still be needed because of the complexity of the problem of determining optimal protection automatically. One of the forms in which this expert user assistance can be implemented (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. Also during the ongoing development of the ACTC such more concrete annotations will be useful, e.g., to test and evaluate individual components being integrated into the tool chain. For this reason, we need to design 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.

Furthermore, we will not only have to develop, test, and evaluate the individual protections integrated into the tool chain during the project, but we will also have to develop, test and evaluate the capabilities of the ADSS to choose the best combinations of protections to protect certain assets. To do so gradually, we will need to be able to specify limited search spaces for the ADSS to explore. So we will need to provide annotations that do not prescribe specific protections to be applied, but 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 might also want to be able to specify simply that a procedure needs to be obfuscated, such that we can study how 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 need more abstract ones. Such more abstract protection prescriptions will, by the way, also be useful in case some expert user assistance proves still to be necessary at the end of the project.

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 [5], 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 [6], 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 a protection specification, an annotation referring to a profile assignment or a security requirement. See Sections 3and 4 for a detailed description of profile assignments and security requirements.

```
<DATA_ANNOTATION> ::= <PROTECTION_ANNOT>
                    | <PROFILE_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>]*)
                    | profile(<ID>)
```

Semantics:

- `protection(<PROTECTION_NAME> [, <PROTECTION_PARAMETERS> ]*)` : protection named `<PROTECTION_NAME>` is applied in the protection scope opened by its `<CODE_ANNOTATION_SCOPE_BEGIN>`.
- `profile(<ID>)` : protections associated to the profile attribute `<ID>` are applied in the protection scope opened by its `<CODE_ANNOTATION_SCOPE_BEGIN>`.

3 Profile Annotations

Section authors:
Roberto Tiella

To provide a flexible way of managing annotations, we introduced a new construct to let developers group multiple occurrences of the same annotation into a common *profile*.

Defining a *profile* is a general method to decouple code annotation from protection techniques actually employed. Source code is annotated with profile annotations which are basically placeholders for actual protection directives that are specified by means of a profile definition file. On running the ACTC, the user can specify by means of a parameter which profile is active and thus which specific protections are applied.

Profile Definition File Profiles are defined in external files. The top-level syntax of such files is as follows:

```
<PROFILE_DECL> ::= profile <ID> { <PROTECTION_ASSIGNMENT>+ }
```

A file can contain many profile declarations. Each profile is a collection of protection assignments. Protection assignments are associations between symbols and protection annotations:

```
<PROTECTION_ASSIGNMENT> ::= <ID> = <PROTECTION_ANNOT> ;
```

Example: In the following snippet two profiles are defined: MILD and STRONG.

```
profile MILD {
       VARS=protection(xor,random);
       STRS=protection(data_to_proc,lutable);
       CRYP=none;
}

profile STRONG {
       VARS=protection(rnc,base(constant(101,103));
       STRS=protection(data_to_proc,lutable);
       CRYPT=protection(wbc, ...);
}
```

The difference between them is that STRONG associates a RNC protection to VARS while MILD associates a xor-masking protection. Furthermore STRONG associates a WBC encryption to CRYPT while MILD doesn't, i.e., "none" is specified instead. Please note that MILD, STRONG, VARS, STRS and CRYPT are just identifiers with no semantic associated a priori.

Using profiles in annotations An externally defined profile can be referenced both in data annotations and code annotations by means of the profile construct:

```
<PROFILE_ANNOT> = profile(<ID>)
```

Semantics: At evaluation time the annotation is replaced with the protection assignment specified by `<ID>` in the active profile. The active profile is a parameter passed to the ACTC.

Example: Suppose a piece of source code is annotated as follows:

```
int x __attribute(ASPIRE("profile(VARS)")))__ = 10;

_Pragma("ASPRIRE begin profile(CRYPT)")

// some cryptographic code here
...
...
_Pragma("ASPRIRE end")
```

and the profiles specification file is the one of the example presented above. If the ACTC is run with STRONG as active profile, the source code is transformed as it would be transformed if annotated with the following protection-specific annotations:

```
int x __attribute(ASPIRE("protection(rnc,base(constant(101,103)))"))__ = 10;

_Pragma("ASPRIRE begin protection(wbc, ...)")

// some cryptographic code here
...
...
_Pragma("ASPRIRE end")
```

If the ACTC is run with MILD as active profile, the source code will be transformed as the following protection-specific annotations were applied:

```
int x __attribute(ASPIRE("protection(xor,mask(random))"))__ = 10;

_Pragma("ASPRIRE begin protection(none)")

// some cryptographic code here
...
...
_Pragma("ASPRIRE end")
```

4 Security Requirements Annotations

Section authors:
Roberto Tiella

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 Deliverable D1.02 [2]:

```
<SECURITY_ATTRIBUTE> ::= confidentiality
                       | privacy
                       | integrity
                       | nonRepudiation
                       | executionCorrectness
```

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.

5 Annotation Facts Storage

Section authors:
Mariano Ceccato

Annotations are embedded in the source code, where they are directly accessible to the source code rewriting tools in the ACTC.

However, code annotations will not survive after the compilation of the rewritten source code into object files. Differently from the source-level part of the ACTC, the binary rewriting part will hence not find annotations in the code to protect.

To solve this problem, the annotations are extracted from the source code files and stored in the form of facts, which are accessible to any component of the ACTC. For this purpose, an annotation extraction module will be implemented (see Section 7.5). It consists of an analysis that takes as input the preprocessed source code, and emits as output the list of all the ASPIRE annotations, formatted in a way that is easy to recognize and to parse by the binary rewriting part of the ACTC (i.e., by Diablo).

For each annotation the following information will be reported:

- **File name:** The name of the source file from which the annotation has been extracted.
- **Line number:** The line of code containing the annotation. In case an interval of code is annotated, the interval is reported (start line, end line).
- **Annotation type:** Indicates whether it concerns a code annotation or a data annotation.
- **Function name:** The function where a code annotation is found, or where a data annotation of a function local variable or parameter is found.
- **Variable name:** The name of the variable that is annotated (only in case of data annotation);
- **Annotation content:** the content of the annotation itself.

The annotation facts are stored in JSON files, with a separated JSON file for each preprocessed file. The name of the JSON file is constructed after the name of the parsed file, by concatenating the extension ".json". An example is presented in Appendix C.

Part II

The ASPIRE Compiler Tool Chain

6 Template for ASPIRE Compiler Tool Chain

Section authors:
Bjorn De Sutter

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 3 shows the way in which different types of tool chain components are visualized.

Secondly, Figure 4 shows the different colors that will be 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 nr. xx, where the following file name extensions are foreseen:
  - **.c**: C source code
  - **.h**: C source code header
  - **.cpp**: C++ source code

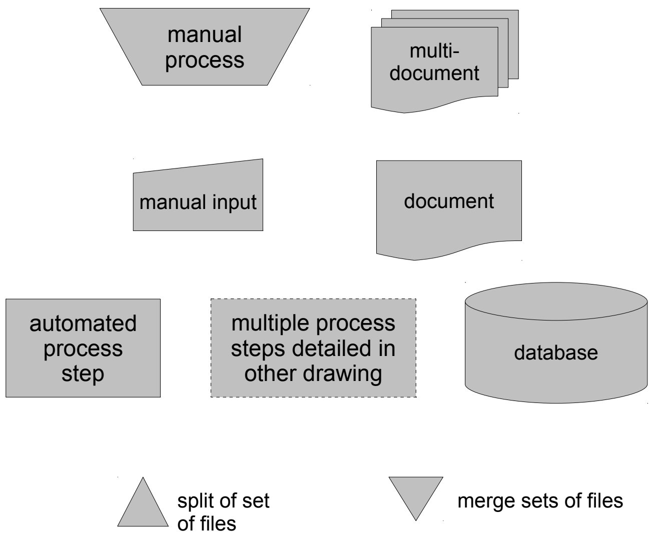

Figure 3: Different types of components in the ACTC tool flow.

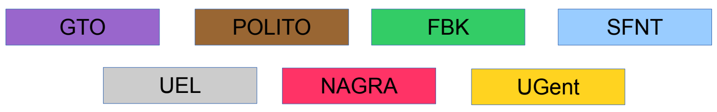

Figure 4: Color codes that indicate the responsible for the concerned components.

  - **.hpp**: C++ source code header
  - **.i**: Preprocessed C source code

- **BCxx**: Binary code document 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

7 The Source-level ASPIRE Compiler Tool Chain in M12

Section authors:
Bjorn De Sutter

Whereas the ASPIRE Compiler Tool Chain is depicted as one monolithic block in Figure 2, it is not at all a single tool in practice. Instead, 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 more refined flow chart in Figure 5.

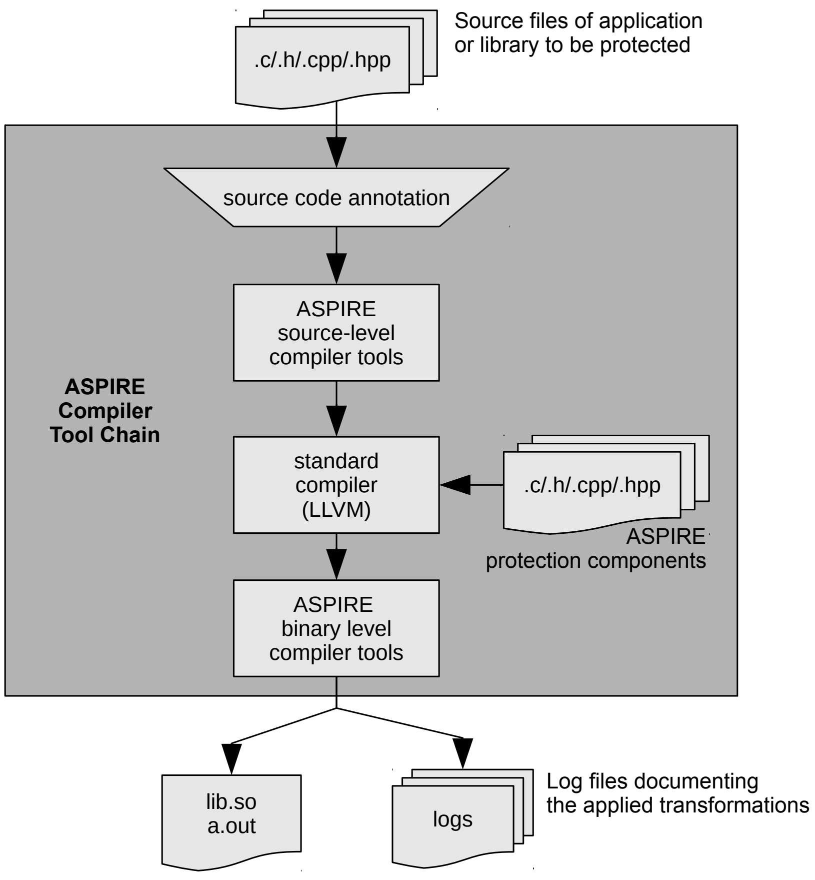
.c/.h/.cpp/.hpp
Source files of application or library to be protected
source code annotation
ASPIRE source-level compiler tools
ASPIRE Compiler Tool Chain
standard compiler (LLVM)
.c/.h/.cpp/.hpp
ASPIRE protection components
ASPIRE binary level compiler tools
lib.so a.out
logs
Log files documenting the applied transformations

Figure 5: 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.
- 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 will 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, will rewrite the binary code in the object files to apply additional protections.

This section presents the design of the source-level components of ACTC envisioned to be integrated in M12 of the project, in time to be demonstrated at the first yearly project review. The binary-level part of the tool flow will be presented in Section 9.

Figure 6 depicts the high-level flow of the source-level ACTC. It includes the manual source code annotation by the user of the ACTC, a preprocessing step for the C code to be protected, a white-box cryptography protection step, and a data hiding step. Furthermore, the annotations are extracted to allow the information contained therein to be passed to the binary-level tools.

In Figure 6 as well as in later flow charts of the ACTC, we have omitted the log files that all tools will 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.

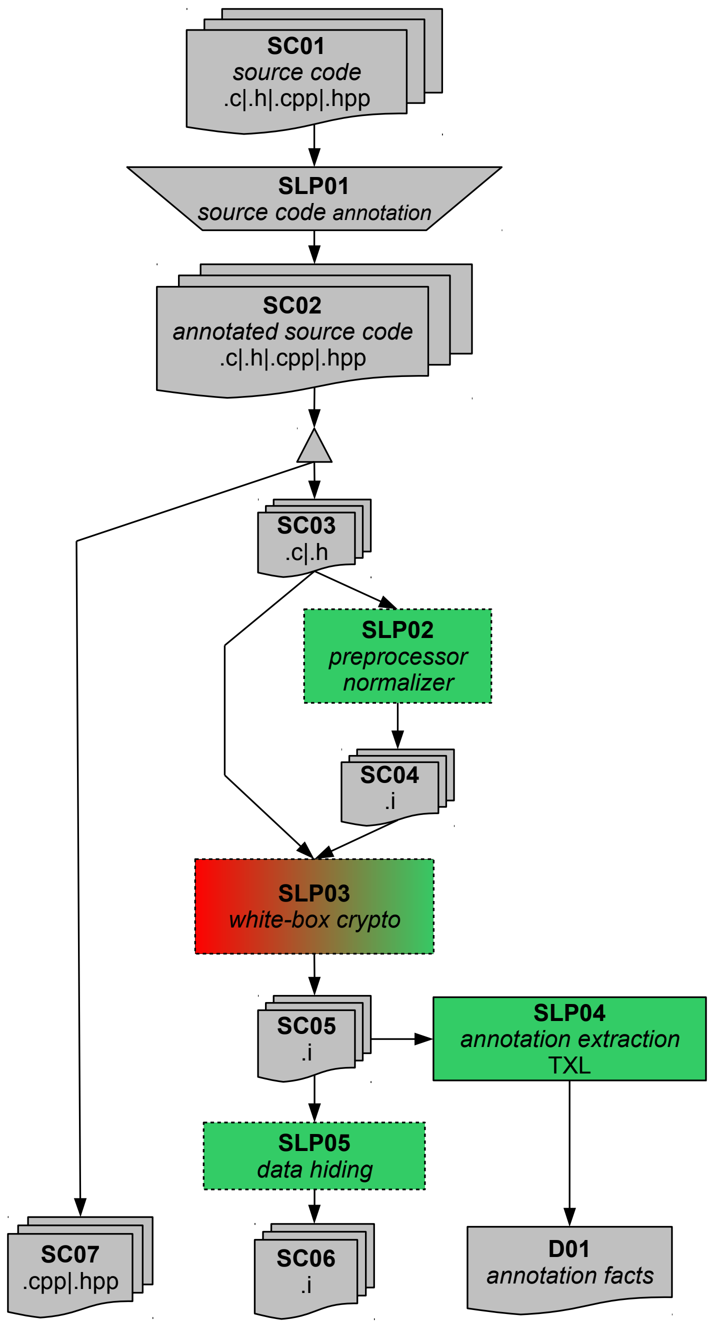

Figure 6: High-level flow chart of the source-level stage of ACTC.

Throughout this document, it is important to keep in mind that the whole ACTC is designed to be modular, such that it will be 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.

7.1 SLP01: Source code annotation

With the annotations described in the Section 2.3, the user first has to annotate his source code (SC01 in Figure 6) manually. In practice, the ASPIRE developers will of course maintain only one code base of their use cases (SC02 in Figure 6), in which they will likely update the code and annotations together.

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

Because the implemented tools will not handle C++ code, the annotated software code base is split in two parts. The C++ part (SC07 in Figure 6) goes directly to the standard compiler, and the C part (SC03 in Figure 6) goes to the actual source-level tools.

7.2 SLP02: Preprocessing - Normalization

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 does not aim for pushing the state of the art with respect to those tasks, so instead we want to reuse 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 will first pre-process all C code in step SLP02.

Figure 7 shows a refinement of this preprocessing/normalization step, in which the first sub-step is the actual preprocessing, which will happen by means of the C preprocessor that is available from the used C compiler. In this figure, components are numbered xx because this preprocessing/normalization step will actually be invoked multiple times, in different places in the source-level part of the ACTC.

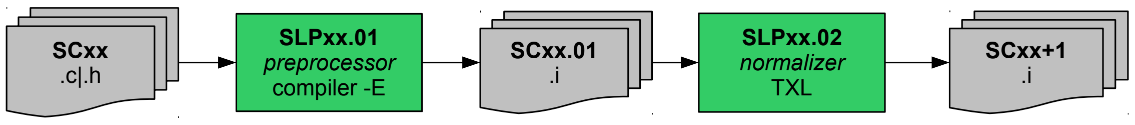

Figure 7: Detailed flow chart of the source-level preprocessing and normalization in ACTC

In the preprocessing step SLPxx.01, one SCxx.01 .i file will be generated per SCxx .c file. The .h headers in SCxx 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 allows 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 will take place and where the ASPIRE consortium wants to push the state of the art.

To do so efficiently, the ASPIRE researchers want to focus as much as possible on protections and protection-specific issues and avoid distraction originating from the intricacies of the C programming language. So where the C syntax allows to express exactly the same semantics with many different syntactic forms, the ASPIRE researchers should not have to spend time on developing support for all those forms.

Consider for example the following six semantically equivalent if-then statements:

```
if (cond) a = b + c;
if (cond) {a = b + c;}
if (cond) a = b + c; else ;
if (cond) a = b + c; else {;}
if (cond) {a = b + c}; else ;
if (cond) {a = b + c}; else {;}
```

To support transformations of all those if-then statements, we have to multiply transformation rules six times to match an if statement. To avoid this, we will apply a normalization step after each preprocessing step, as depicted in Figure 7. With normalization, we reduce all the variants to a canonical form. Transformation rules then need to consider just the canonical variant, so they can be much simpler.

As the project proceeds and support for more complex transformations is implemented in the source code transformation tools (see below), the normalization transformations will be developed accordingly.

The normalizer will be implemented using the TXL programming language and its free development tools (http://www.txl.ca/). FBK is responsible for the development of the preprocessing/normalization step, and for its integration in the ACTC.

7.3 Ordering the Source-Level Protections

After the code has been pre-processed and normalized, 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. White-box cryptography, data obfuscations, and client-server code splitting are the ones to be developed first in the project. The question therefore then arises in which order to apply these transformations.

During a number of conference calls with all consortium partners, we have decided that the most appropriate order is the following:

1. white-box cryptography (see deliverables D1.04 Section 3.5 and D2.01);
2. data hiding (see deliverable D2.01);
3. client-server code splitting (see deliverables D1.04 Section 3.3 and D3.01).

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.

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.

7.4 SLP03: White-Box Cryptography Protection

Figure 8 depicts the detailed compilation tool structure of the first protection deployed on source code in the ACTC: white-box cryptography. We refer to D2.01 for a detailed description of the transformations that need to be applied, and to D1.04 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 SC04. As discussed in Section 1, in this phase of the project, 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 will be described in more detail in D2.04.

This configuration file is an XML file that specifies the requested transformations by means of key-value pairs. This format was chosen because NAGRA's existing tool is already based on XML configuration files.

[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.

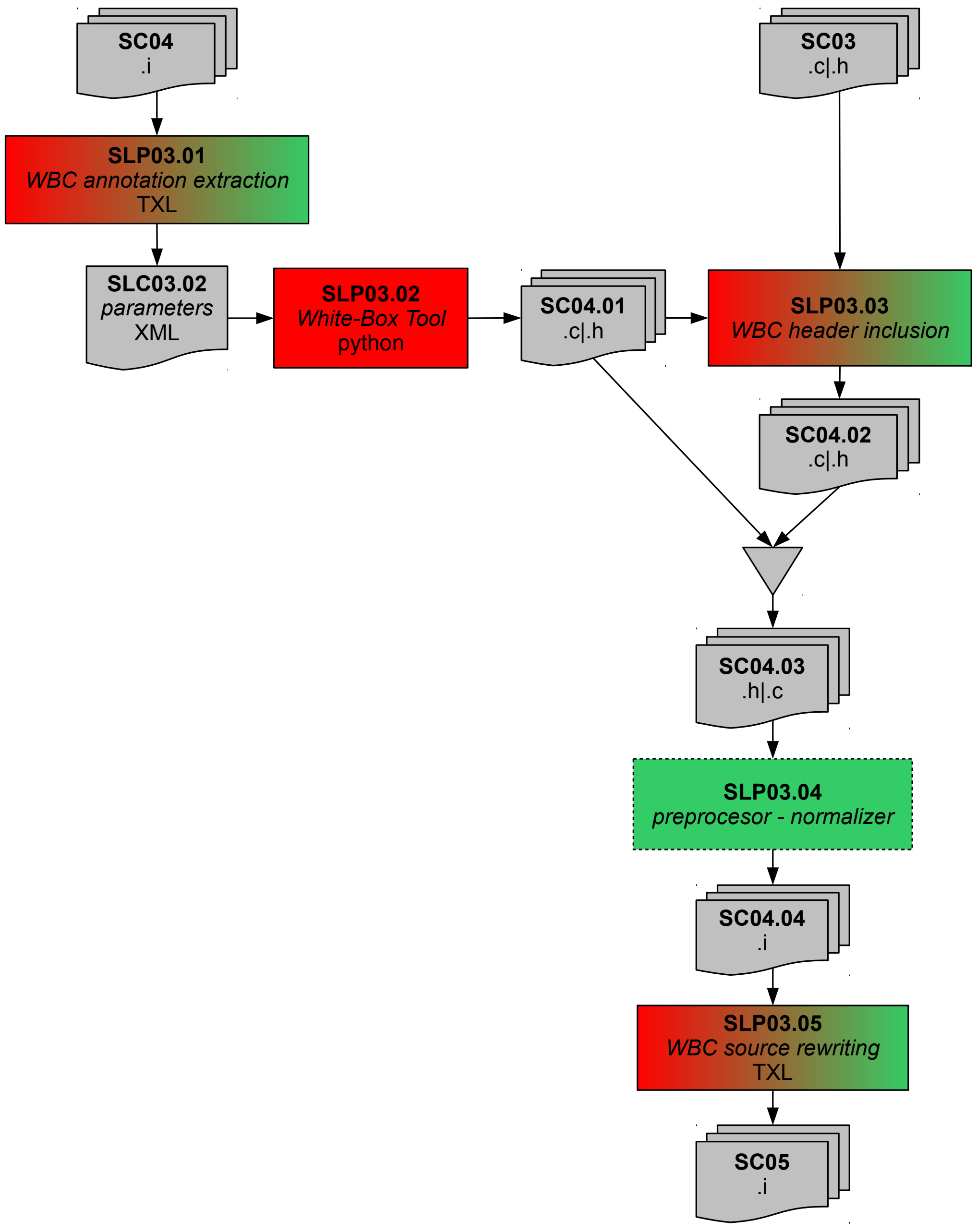

Figure 8: Detailed flow chart of the white-box cryptography tool flow in the ACTC

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 and normalized, hence the additional rewriting step SLP03.04, which re-performs the steps already discussed in Section 7.2.

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 SC05. In this step, based again on TXL, procedure calls and key values are replaced, as will be described in detail in D2.01 and D2.04.

7.5 SLP04: Annotation Extraction

Now that all the application code and the white-box cryptography primitives are available for further analysis and transformations, component SLP04 can extract all the annotations from the source code that will drive the binary-level part of the ACTC.

All information encoded in the annotations will be extracted and stored in a document D01. For each annotation, its line number information will be included, as well as some other auxiliary information that helps in identifying the annotated code. This information will, e.g., include the file name, the procedure name, the variable name, etc.

For the first iteration of the ACTC, only the file name and procedure names are relevant, because those are the only identifiers that the binary-level tools based on Diablo can handle. In later versions of the tool flow, the line number information will be used for more precise identification of annotated code fragments. That line number information will then be related to code addresses in the binary code by means of the debugging information that the standard compiler will insert into the object files, and that Diablo will extract again.[2]

As line number information is not yet needed in M12, there is also no need to maintain line number information yet. However, in later versions of the tool flow, such bookkeeping is foreseen. In particular, when lines numbers in the source code are affected by source-to-source transformation, it is necessary to update the extracted line numbers of the annotations. This will be necessary to (1) pass the correct line numbers to the binary-level tools, (2) present correct line numbers (as in the original application code SC02) to the user of the ACTC in the logs and reports that each tool will produce.

7.6 SLP05: Data Hiding Transformations

Figure 9 depicts the final step of the source-level ACTC. Besides the code transformation pass SLP05.02, which will again be implemented using TXL, this step features an advanced code analysis pass in SLP05.01, which will be implemented by means of GrammaTech CodeSurfer.

Whereas inserting invocations to white-box crypto primitives is a relatively simple, local transfor-

[2]Limited support for reading and handling debug information is already available in Diablo, but it needs to be extended to handle the debug information versions as produced by the binutils assembler that will be used in the ACTC.

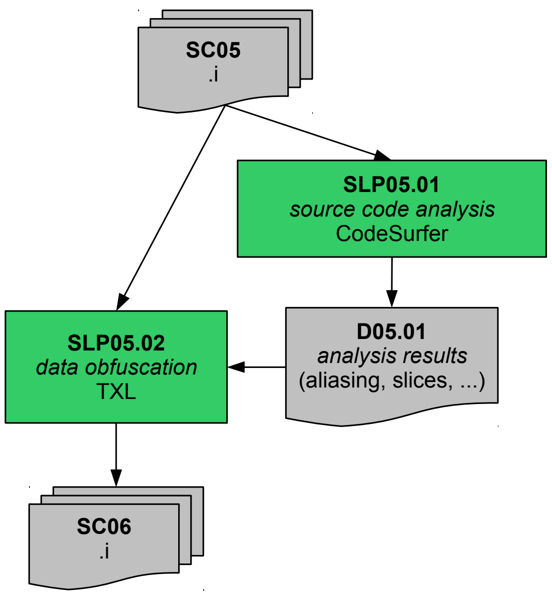

Figure 9: Detailed flow chart of the data hiding components in the ACTC

mation, data hiding transformations are not so local.

When a decision is made to hide the value stored in some variable by encoding it using a special encoding not know to attackers, encoding operations need to be inserted wherever a non-encoded (or a differently encoded) value is to be stored in the variable, and a decoding operation needs to be inserted wherever the variable is read and its value is to be used in non-encoded form. This will be explained in more detail in D2.01.

A data flow analysis is needed to decide where to insert these operations, and also to check whether it is possible at all to apply a data hiding protection to a variable: when alias analysis cannot guarantee that a variable will only be accessed by a limited set of read and write operations that can all be rewritten, it is not safe to apply a transformation. GrammaTech's CodeSurfer provides a framework on top of which custom data flow analyses can be implemented. For the data hiding protections, and later also for the client-server code splitting, custom analyses will be developed that compute the required data flow information in SLP05.01.

The data hiding source code rewriter will then rewrite the software as requested by the source code annotations (see Section A) and to the extent allowed by the data flow analysis results.

In later versions of the tool flow, the analyses will be extended to analyze to what extent data hiding annotations can be propagated throughout the program. For a simple example, suppose there is a function `int add(int x, int y)` that simply adds the values of `x` and `y` and returns the sum. Then consider a code fragment

```
int x __attribute__((ASPIRE("protection(xor,mask(constant(12)))"))) a = 5;
int x __attribute__((ASPIRE("protection(xor,mask(constant(12)))"))) b = 6;
int x __attribute__((ASPIRE("protection(xor,mask(constant(12)))"))) c = add(a,b);
```

Ideally, this fragment should not be rewritten into

```
int a = 5^12;
int b = 6^12;
int c = add(a^12,b^12)^12;
```

but instead it should become something along the following lines:

```
int a = 5^12;
int b = 6^12;
int c = add_encoded(a,b);
```

in which `add_encoded()` is a rewritten version of `add()` that operates on encoded values.

8 Compiler, Assembler and Linker

Section authors:
Bjorn De Sutter

8.1 Compiler Requirements

The source code files SC06 of the application protected at the source level will be compiled by a "regular" compiler. The generated assembler files will then be assembled by a "regular" assembler, and the generated object files will be 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 will be used in the binary-level protection tools in ASPIRE. 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 data correctly yet. Complete support for exception handling is foreseen for late 2014. In the mean time, CLANG-LLVM and GCC need to be invoked with the flags `-Wl,--no-merge-exidx-entries`.

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

8.2 Compilation and Linking Tool Chain

Figure 10 depicts the use of a regular compiler, assembler and linker in the ACTC. All source code files are compiled with Clang - LLVM 3.4 to produce .s assembly files (SC08). These are assembled

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 for late 2014)
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

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 linker is also invoked with the -Map flag to produce a so-called map file (D02) that logs the operation of the linker.

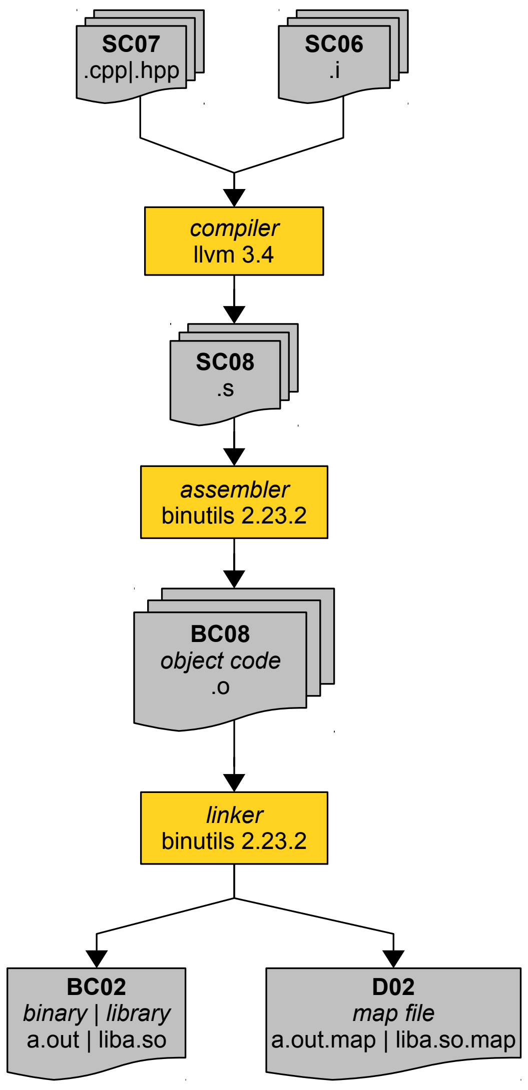

Figure 10: Compiler part of the ACTC.

9 Binary Rewriting Tool Chain

Section authors:
Bjorn De Sutter

9.1 Overall Binary Rewriting Approach

The overall binary code protection approach in ASPIRE consists of four major steps, as depicted in Figure 11.

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.

For example, for the client-side code splitting by means of an embedded SoftVM (see D1.04 Section 3.1), bytecode needs to generated to replace native code sequences in the protected application, and this bytecode needs to be linked into the application. Also the SoftVM itself needs to be linked into the application, and in the more advanced version of the client-side code splitting, the SoftVM internals will be customized for the application in which it will be embedded. Before generating the bytecode and the customized VM, the code to be protected needs to be analyzed, and decisions need to be made about which native code will be replaced by bytecode. The necessary analyses

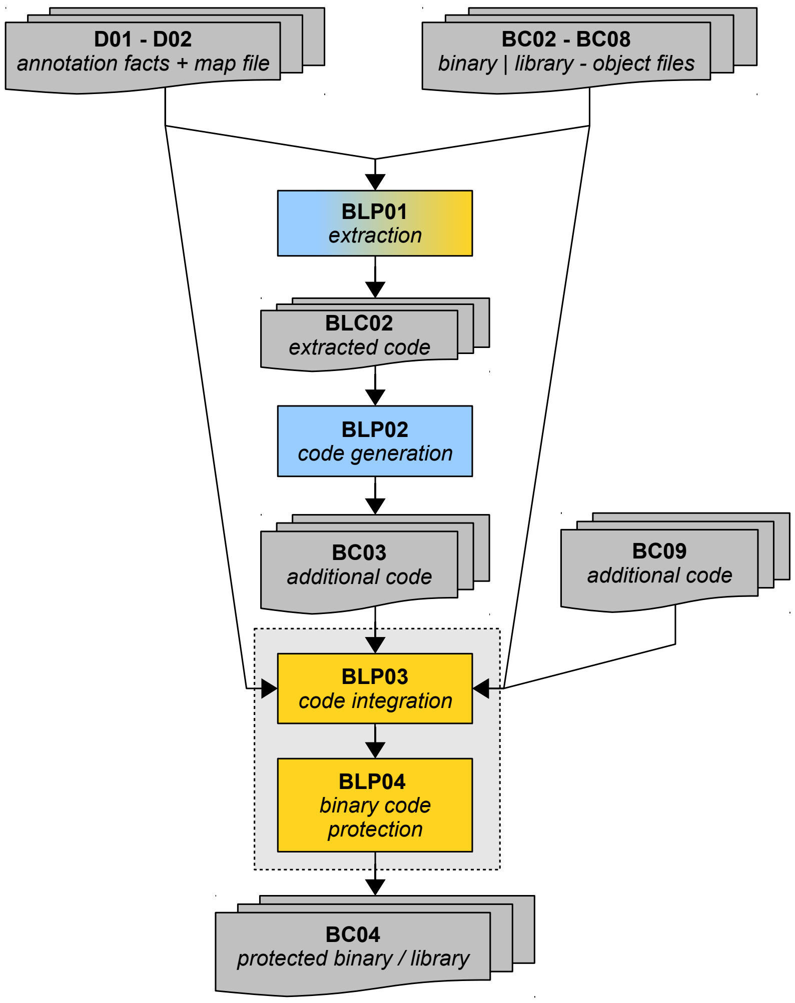

Figure 11: Four steps of the binary-level part of the ACTC

and decisions are part of the first step, the result of which is a set of configuration files to drive the components that will generate the custom software components.

This generation of custom components in the form of object files BC03 constitutes the second step BLP02 of the binary-level ACTC.

In the third step, BLP03, the custom components of BC03 are integrated: they are compiled and linked in into the application, toghether with other, fixed software components BC09 that are also part of the protection, but that were compiled separately and independently of the preceding parts of the ACTC because those fixed components BC09 do not need to be customized for the application at hand.

Furthermore, the original code of the application is rewritten to actually invoke the custom components as needed. For example, 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. This integration will happen protection-per-protection.

In the fourth step, BLP04, all rewritten code and all integrated components are further protected by applying obfuscation and anti-tampering protections. Furthermore, the final code layout is determined, and the code is assembled and relocated (when necessary), such that the final binary code is finally known. At that point, place-holders that might have been inserted during the rewriting in steps three and four can be filled in.

In practice, most of BLP03 and BLP04 will be performed during one invocation of a binary rewriting tool based on the Diablo link-time rewriting framework. For that reason, we have grouped these steps in Figure 11.

9.2 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 12 years. For different purposes (such as speed optimization, code compaction, code protection), different front-end tools have been built on top of this framework.

9.2.1 Basic Diablo Operation

Figure 12 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, 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.

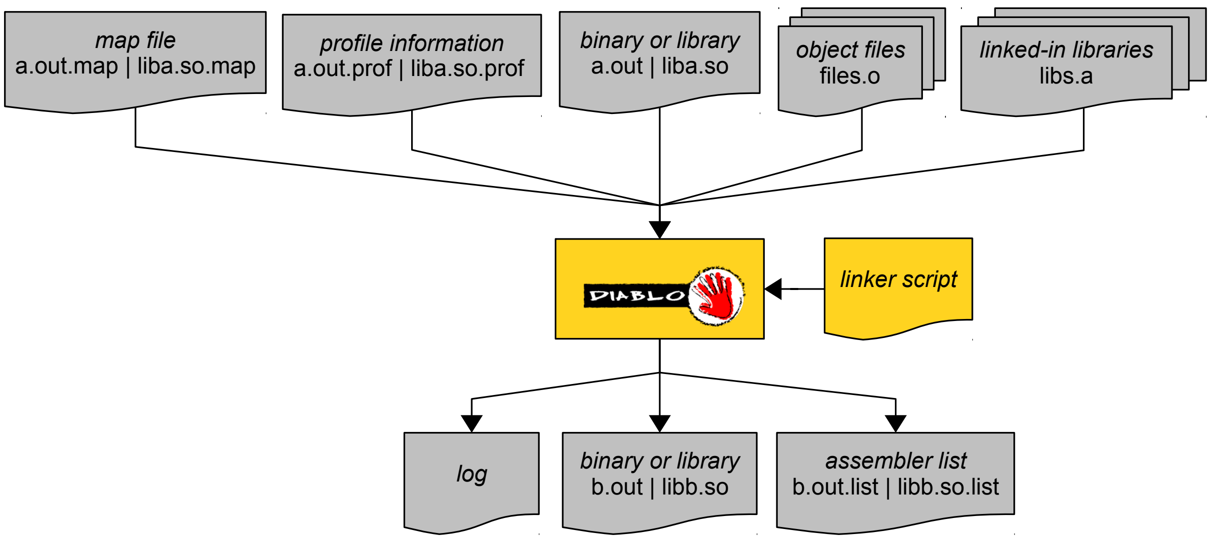

Figure 12: Inputs and outputs of Diablo.

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.

9.2.2 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.

9.3 Client-Side Code Splitting

Client-Side code splitting (D1.04 Section 3.1) is one of the protections that will be implemented in the three first steps as discussed in Section 9.1. In its initial implementation, a fixed SoftVM, which requires no customization, will be embedded in the code to protect,

9.3.1 BLP01: Native Code Extraction

As indicated in Figure 13, in BLP01.01 a Diablo rewriter will collect 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) and the object code (BC08) that was linked into the original application by the standard linker.

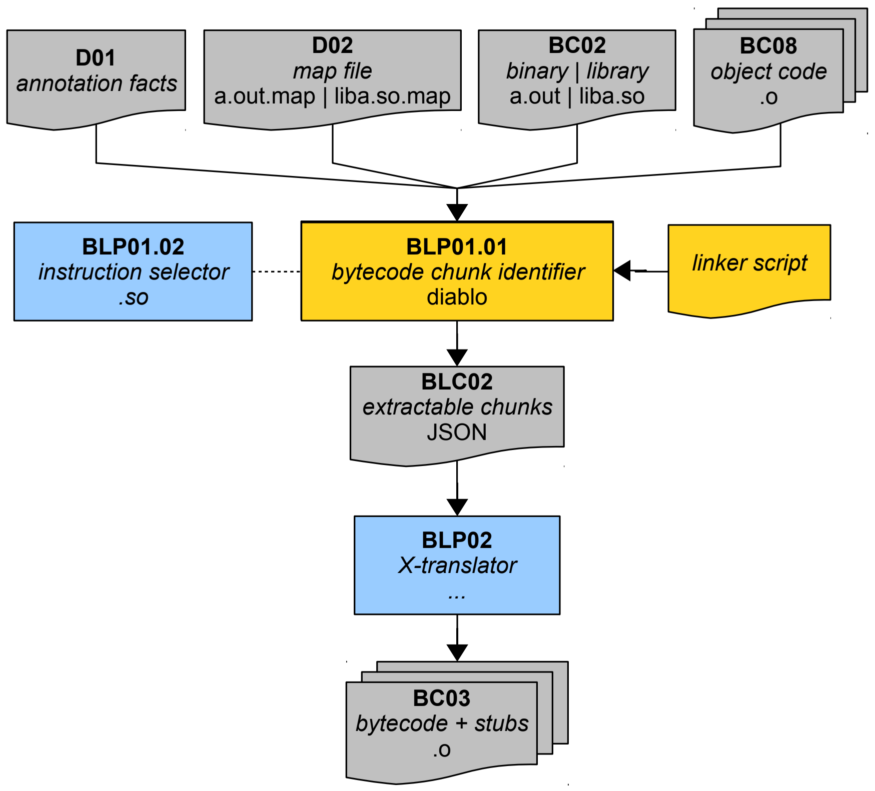

Figure 13: Tool flow components for chunk extraction and bytecode generation

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

To select the native code fragments to be translated to bytecode, the Diablo tool will consider procedures marked as such in the annotation facts D01. Within these fragments, all possible fragments will 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.

9.3.2 BLP02: Bytecode Generation

The second tool BLP02 in support of client-side code splitting is the X-translator. Based on the JSON files of BLC02 it generates bytecode, as well as stubs that will replace the selected native code fragments. The responsability 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, and back after 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 is responsible for the code extraction in the Diablo rewriter, and SFNT is 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.

However, unless special care is taken, this separation of concerns could introduce some (unwanted) dependencies between the involved partners' tools. Over time, the subset of the ARMv7 instruction set that is supported by the X-translator and the SoftVM will grow. So over time, the code fragments to be selected by the Diablo rewriter will grow. To avoid the need to keep the three tools spread over two partners synchronized with respect to the supported instruction set, we have decided to lift that responsibility from the Diablo rewriter, and to move it into a small dynamically linked library BLP01.02, the so-called instruction selector in Figure 13, that will be maintained by SFNT, and that will be invoked by UGent's Diablo rewriter to select the instructions that can be translated to bytecode.

9.4 BLP03: Code Integration

In the initial implementation of the ACTC, with the fixed SoftVM, the integration of the generated bytecode and code stubs, as well as of the SoftVM itself is straightforward. First, as shown in Figure 14, the SoftVM source code SC09 is compiled with the same tool chain used to compile the code to be protected. The result of this compilation process consist of the SoftVM object code files BC09.

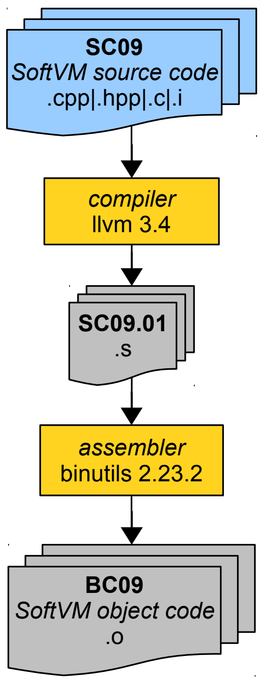

Figure 14: Compilation of SoftVM

Next, the originally generated object code BC08 of the original application protected at source level (see Figure 10), is relinked with the generated bytecode and the stubs (BC03), and with the

SoftVM object code (BC09), as depicted in Figure 15. This produces a new application BC04, with the corresponding map file D04. We use the names c.out and libc.so[3] to indicate that these files denote extended version of the original binary a.out or the original library liba.so of BC02.

Finally, Figure 16 shows the last step, in which a second tool BLP03 based on Diablo will rewrite that application to finalize the protection. This tool will replace the native code fragments that have been translated by the X-translator in step 2 by control flow transfers to their corresponding stubs. UGent is reponsible for all of this integration in the Diablo tool.

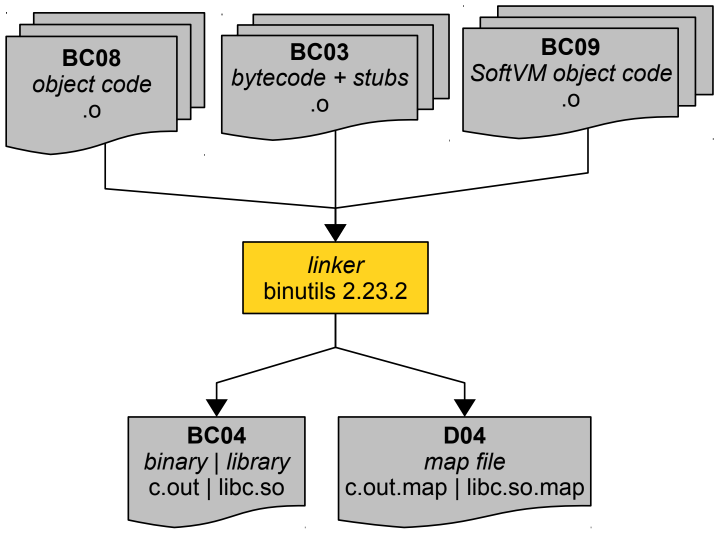

Figure 15: Linking of the SoftVM

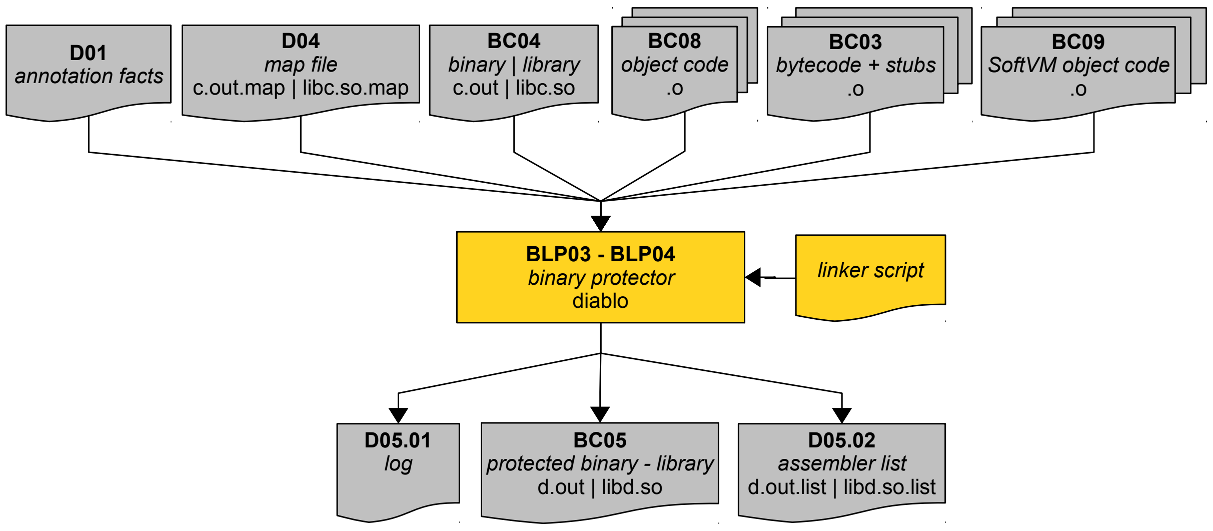

Figure 16: Integration of the SoftVM and application of binary code obfuscation.

9.5 BLP04: Binary Code Control Flow Obfuscation

In the same run of the Diablo tool BLP03, as depicted in Figure 16 control flow obfuscations will then be applied in step BLP04. These obfuscations include the insertion of opaque predicates, the flattening of control flow, and the insertion of branch functions. They will be applied pseudo-

[3]The file names b.out and libb.so were already used for naming the output of Diablo in general in Figure 12, so we do not reuse them here to avoid confusion.

randomly in procedures annotated as candidates for these protections according to the annotation facts in D01.

On top, (part of) the stubs inserted for client-side code splitting will be inlined, code factoring will be performed to couple the code from the original application and the code of the SoftVM more tightly, and the code layout will be randomized. In combination, these transformations will ensure that the original code and the SoftVM code are intertwined.

UGent is reponsible for all of these transformations.

The end result will be a protected application BC05, a log of the applied transformations D05.01, and an annotated assembler listing D05.02. We use the names d.out and libd.so in Figure 16 to mark that they correspond to rewritten versions of c.out and libc.so.

10 The ASPIRE Source Code Sanity Checker

Section authors:
Andrea Avancini

The ASPIRE Source Code Sanity Checker is a basic tool to test and quickly evaluate if source code that is intended to be the subject of ASPIRE protections can be handled by the ACTC, and more particular by the ACTC components that build on the TXL programming language and Grammatech CodeSurfer. Both are extensively used in the source-level part of the ACTC.

The Sanity Checker simulates the execution of one of the steps in the source-level ACTC as depicted in Figure 6. The checker performs the preprocessing of source files, applies analyses with CodeSurfer and TXL, and generates the binary for the target application. As input, ASPIRE Annotated code is supported.

TXL and CodeSurfer are both used to extract pieces of information from the source code under check. The outputs of CodeSurfer and TXL analyses are compared to identify possible discrepancies in the application of the two tools. If the information retrieved is identical, the execution of the Checker is considered regular and the application under analysis is supposed compatible with the ASPIRE source level tool chain. In case of differences in the analysis results or other errors, the original source code is flagged as potentially non compatible.

The Sanity Checker can give useful information on the suitability of the ASPIRE tool chain for the checked code to be protected and on the possible problems that can manifest at early stages. The checker can be used by the use case and toy example developers in ASPIRE to perform verification on their code, as well as by the developers of analyses on top of CodeSurfer and TXL to test their tools. Users are given error messages in case the tools are not able to handle the source code: common issues at this level can be related to parsing errors, such as unsupported C code by the TXL grammar or errors related to CodeSurfer's internal parser; or to the size of the code to protect, when the original application is too large to be handled by the tools.

The main objective of the sanity checker is to assist and facilitate the development of the ASPIRE use cases. However, in future evolutions of the ACTC, the sanity checker might be integrated as the very first step in the tool flow, to assure that the code to be protected is compatible with the adopted tools.

11 The ASPIRE Shared Build Environment

Section authors:
Bart Coppens, Bjorn De Sutter

To ensure that partners can test their contributions to the ACTC and to the ASPIRE demonstration in the exact same, shared build enviroment, 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 8, as well as all other components of the ACTC described in this document.

The patched compiler tool chains and the ACTC components 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. Currently, 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 controled package repository, this build environment will enable all partners to create reproducible builds. This implies that we can, at any point in time, roll back to previous versions of the compiler tool chains and the other ACTC components.

The VM image, which can be run with both VirtualBox and VMWare, is based on a minimalistic 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.

Currently, 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.
- A patched compiler tool chain. Currently, the version distributed from the ASPIRE repository contains a patched gcc 4.8.1 with binutils 2.22a, and eglibc 2.17. Shortly, LLVM 3.4 will be added.
- Up-to-date versions of Diablo from the ASPIRE repository.
- 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.

Users of this VM will be able to use the standard Debian package management tools to update their build environment VM to the latest version of ACTC components as available in the ASPIRE repository.

12 Ongoing Design of Future ACTC Versions

Section authors:
Bjorn De Sutter

Besides the short term design described in the previous sections, which serves as a guidance document for the implementation of the M12 ASPIRE ACTC, we are also working on the longer-term design of the tool flow. That current results and ongoing activities are described in this Section.

12.1 Integration of Mid-term Protections

For the next batch of protections that will need to be integrated into the ACTC, we already have a concrete vision on where to integrate them in the existing tool flow:

Client-server Code Splitting As already described in Section 7.3 this protection (see D1.04 Section 3.3) will be implemented after the data hiding step SLP05 (Figure 6) in the source-level tool flow. This new step will be structured very similarly to the data hiding tool flow as depicted in Figure 9, except that it will also produce code to be executed on the ASPIRE security server.

Multi-threaded Cryptography (see D1.04 Section 3.6) can be implemented at any stage following the WBC protection in the source level tool flow. A similar tool flow as those for data hiding and the above protection will definitely suffice, as this protection is quite mechanistic, requiring little code analysis.

Anti-Debugging (see D1.04 Section 3.2) will be implemented similarly to the client-side code splitting step described in Section 9.3, in a step immediately following the client-side code splitting. Extraction of the code to be transferred from the application process to the debugger process, as well as all necessary rewriting will happen right after the SoftVM code is integrated, such that all code, including SoftVM code and stubs, can potentially be moved into the debugger process.

Code Mobility (see D1.04 Section 3.4) will definitely be implemented in Diablo after the application of binary control flow obfuscations. The reason is that whether or not some code fragment is mobile or static, we want to be able to protect it with the other, already integrated or aforementioned protections as well. So any code extracted from the client-side application or binary to become mobile already needs to be transformed first. Furthermore, applying the code mobility protection late will also allow us to make the SoftVM code itself mobile.

With respect to the SoftVM-based protection, we also foresee an additional post-pass at the very end of the binary rewriting process. In this post-pass, Diablo will collect the return addresses where control has to be returned to after the bytecode fragments have been interpreted by the SoftVM. These addresses will then be passed to an external tool that encodes the addresses in a way that the SoftVM can decode. The encoded addresses will then be replace placeholders in the binary. The reasons for storing them in an encoded form is of course to prevent simple static attacks from extracting the return address.

This described post-pass process in essence implements a form of relocation process. Like the X-translator and the instruction selector component in Figure 13, this external relocation tool will be maintained by the owners of the X-translator and the SoftVM. This will facilitate the development of different encodings over time, and eventually renewability.

For other protections, including most of the anti-tampering techniques described in Section 4 of D1.04, more research is needed and ongoing before we can decide exactly how to integrate them into the ACTC.

12.2 Composability and Protection Optimization Process

In the previous sections, we described the order in which the different protections will be applied. In order to really integrate them in a composable way as foreseen in D1.04, we need to consider much more features of the individual protections, such as preconditions and postconditions for applying the protections.

Furthermore, eventually the ADSS will have to determine the protections to be applied, with the help of the analyses available in the ACTC. As described in Section 1.2, choosing a good design point to split the responsibilities and decision process between the ACTC tools and the ADSS is a complex task. One of its goals is to shield the ADSS from all too detailed information.

It is clear that the aforementioned two tasks of composable protection integration into the ACTC, and integration of the ADSS and ACTC require the collection of much more detailed information about the individual protections and their corresponding compiler transformations than what is described in this document.

To that end, a set of documents, one per protection, has been assembled and is being updated and extended to collect all the necessary information. These documents instantiate an extensive template to ensure that all partners are well aware of all relevant aspects, and document their insights on those aspects as soon as possible during the project.

The collection of all documents is then used to build, through brainstorms and discussions under the guidance of the project coordinator, a vision for the design of the final, integrated ACTC and ADSS. In support of the ADSS, the pre-conditions and post-conditions will later also be formally modelled into the ASPIRE security model.

12.3 Integration of Profiling Information

In addition to letting the ADSS optimize the achieved protection, the ADSS will also have to consider the impact of the protections on performance. In order to do that, profile information is needed to estimate the frequency with which individual code fragments are executed and their contribution to the overall execution time. For that purpose, a prepass will be added to the ACTC in which Diablo is used to produce an instrumented binary or library version. By invoking this instrumented version on a set of representative inputs, profile information will be collected. All in all, this is a fairly trivial extension. The support for this profiling in Diablo is developed by UGent. Apart from more extensive testing and debugging, it is ready.

Furthermore, to compute metrics on the code before and after transformation as needed to evaluate the impact of protections on potential attacks, additional profile information will be collected as needed by the concrete metrics that will be documented in deliverable D4.02. This information will include complete traces of all executed instructions and of all memory accesses, as well as the variability observed in individual instructions' behavior. For example, for certain metrics we will want to know whether an instruction always computes the same value or not, and whether a conditional branch instruction is always taken or not, in order to estimate the amount of information that an attacker can get from using some of the dynamic attack techniques discussed in D1.02 Sections 4.4.2, 4.4.3, and 4.4.4. The necessary profiling support for collecting this information still needs to be developed by UGent in Diablo.

Part III

The ASPIRE Decision Support System

This project aims at using several lines of defence to protect applications. Each technique has its pros, cons, peculiarities, technicalities, pre- and post-conditions, hence its correct deployment usually demands a software security expert (team). This is usually needed since traditional software developers are typically not proficient and not trained in the software protection area. This is certainly a cost for companies that have to find skilled people. Moreover, the protection phase increases the time-to-market of new applications and versions.

Furthermore, the situation is made harder because, in order to find the most suitable set of protection techniques to defend an asset, experts must take into account a great number of parameters for each protection technique and manage relationships between protections. Some protection techniques are mutually exclusive or need to be deployed in a specific order to coexist. Others can interfere or, when used together, can dramatically increase performance overhead. Understanding potentials and limitations of protections composability is one of the open issues we are addressing in this project.

The main problem the ASPIRE Decision Support System (ADSS) wants to address is determining (semi-)automatically the most suitable set of protection techniques that protect the application assets and that satisfy a set of user constraints. This problem will be referred to as *ASPIRE decision problem*. The ASPIRE decision problem can be constrained in several ways, e.g., by arbitrarily selecting protection techniques to use, on the expected protection level, on application performance and network bandwidth consumption and overall network traffic. Like many other optimization problems, the ASPIRE decision problem is a problem whose solution exceeds human capabilities and it is thus manually performed by experts that use their expertise and experience to find reasonable tradeoffs.

Still, optimization problems can be addressed in a computer-aided manner. ASPIRE proposes a (semi-)automatic system that will assist developers in protecting their application, i.e., in solving the ASPIRE decision problem, with the final goal to reduce the human factor in such delicate decisions, and to eliminate it altogether on the long term. In this context, the ASPIRE project will develop an expert system, the ADSS that will enable also software developers without specific knowledge on software protection to protect a software application by (semi-)automatically choosing the most adequate combination of protection techniques and their configuration parameters, and by steering the ACTC to apply the selected protections.

13 A bird's eye overview of the ADSS

Section authors:
Cataldo Basile, Daniele Canavese, Rachid Ouchary

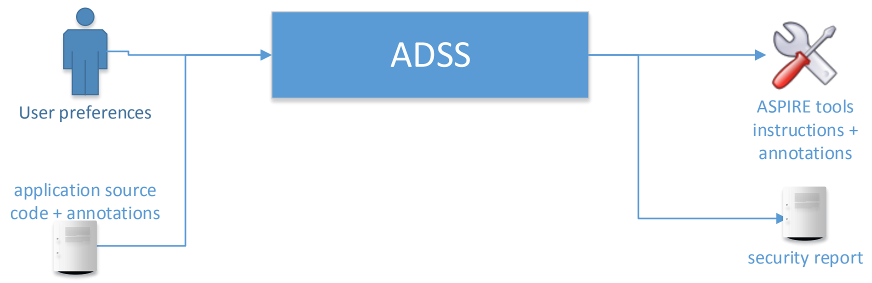
ADSS
User preferences
application source code + annotations
ASPIRE tools instructions + annotations
security report

Figure 17: The ADSS inputs and outputs.

Figure 17 sketches the inputs and the outputs of the ADSS. The typical usage of the ADSS, as was already explained in the deliverable D4.01, consists of the following main steps[4]:

1. The user, whether he is a protection expert or not, passes to the ADSS the source code of the application to protect. However, he must also instruct the ADSS by choosing what to protect and, possibly, giving some hints on how to protect it. That is, the user must specify the security requirements, which include also identifying the assets in the application to protect. Security requirements will be described by means of annotations, added in the application source code by the application developers to tag the portion of code he wants the protection methods to focus on and to mark the assets, according to the annotation syntax as already specified in Section 2.

2. The user may provide additional preferences to constrain the decision process, such as the protection techniques not to use, acceptable overhead, the needed level of protections, the type of attackers/attacks that need to be considered, and possibly other relevant information about the platform, the business model of the software developer, assumed online connectivity, server requirements, etc. This information can be provided by means of GUIs, as annotations are not designed to integrate these ADSS constraints either. Depending on the inference capabilities we will be able to implement during the project, the ADSS will be able to identify when there is some missing information and ask focused questions to the user when all the needed data to start the automated process are not available.

3. The ADSS determines the best combination of protections techniques that can be applied to protect the application assets by solving the ASPIRE decision problem. In the following sections, we will denote the best combination of protections selected by the ADSS as *golden combination*.

   For this task, the ADSS uses the information stored in the ASPIRE knowledge base (AKB). The AKB initially contains the a priori knowledge[5], which is independent of the application to protect and it is then populated by the ADSS with application-specific information. Application-specific information includes extracted annotations as well as the other data

[4] All these steps will be elaborated better in next sections.

[5] Introduced in D4.01, the a priori knowledge comprises information that is independent from the program to protect. It models attacks, attack categories, expertise/skill/resources/ needed, software protections, their relations, the possibility to use them in combination to strengthen the overall protection, assets, threats, and types of security properties to protect in applications. Further details in D4.01 Section 3.1.

useful for the decision process extracted by a set of analysis tools. Moreover, on the basis of the collected application-specific knowledge, the AKB is extended by means of an ad hoc enrichment framework and of the attack simulation models (ASMs) presented in D4.01. The result is the execution-specific knowledge. One of the main goals of this part is to explain how the ADSS will perform this enrichment task.

4. A detailed security report is presented to the user listing the protections the ADSS has chosen, the actual parameters, and the estimated effects on the application to protect. At this point, the user can decide to modify the initial security requirements and constraints and repeat the process until he is satisfied with the obtained results.

5. The ADSS instructs the ASPIRE tool chain on how to protect the application to enforce the golden combination and releases the control. This is achieved by by producing instructions for the ACTC in form of protection-specific annotations or as a set of additional data that are directly usable by the ACTC.

It is worth noting that during the actual deployment of the golden combination something could fail. For instance, a technique can not be actually deployed on the target application or there is some incompatibility between protections that have not been foreseen by the theoretical model. In that case, the user is required to restart from step 1. The information acquired during the ACTC execution can be made available to the ADSS to learn how to the same incompatibility issue both when trying again to protect the same application and possibly for other applications. This is easily done by updating the acquired knowledge in the AKB.

Moreover, (possibly expert) users can also take into account the differences between the ADSS' estimate of what protections would be applied (described in the produced report), and the actually applied protections as reported by the ACTC (and possibly even as observed through analysis of the generated code). This form of discrepancy analysis is certainly useful for the ASPIRE Consortium for debugging purposes, it is needed to allow users to have more control on the process and decide if the ADSS need to be executed again. But this form of double check is also needed to reassure experts on what the ADSS and ACTC are doing on their applications and give confidence on our results.

13.1 ADSS workflow

In order to select the golden combination, the ADSS must perform several operations. First, the ADSS must identify the suitable protections. Next, the ADSS must evaluate how suitable protections can be combined to strengthen the protection level. The result of this step will be *suitable combinations*, that will form the *solution space* (of the ASPIRE decision problem). The golden combination will be selected in the solution space by means of an *optimization process*. The optimization process will be split in two phases: first the *optimization program* will be generated, then the optimization program will be solved with off-the-shelf *solvers* (like lp_solve or CPLEX). Finally, the golden combination selected by the solver will be processed to obtain the instructions for the tool chain and the annotations. If the identified suitable combinations are too many, the solution space can be too large. A large solution space can negatively affect the performance of the optimization process. Therefore, an optional pruning process can be performed to reduce the solution space.

The optimization program includes a target function, a.k.a. the fitness function, that needs to be minimized or maximized, and several constraints. To perform this task, the ADSS uses metrics and other results from WP4. To evaluate the target function, the suitable combinations (which are the independent variables in the optimization program), and the protections they are composed of, are evaluated according to several criteria (protection level, overhead, etc.) by means of functions that map them to integer or real values used by the target function to select the golden combination. According to the current draft of D4.02, the suitable combinations will be evaluated by considering several *measurable features*. Measurable features are application properties that can be correlated to protections and attacks. The relations (i) between measurable features and protections and (ii) between measurable features and attacks will be discussed in D4.02.

At this point, we can introduce the general execution work-flow performed by the ADSS, as visualized in Figure 18. All the phases shown in Figure 18 use the AKB as the main repository: the AKB is where all data are stored on and retrieved by each work-flow phase. We recall that before the ADSS workflow starts, the AKB contains already the *a priori knowledge*[6].

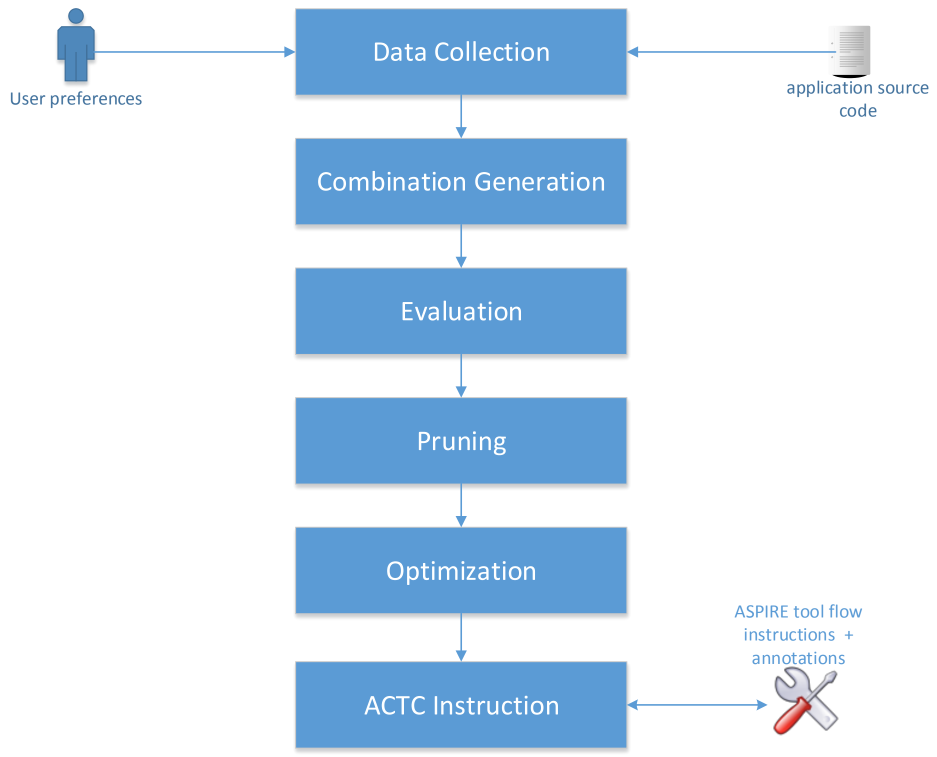

Figure 18: The general ADSS work-flow.

13.1.1 Data Collection: obtaining the application-specific knowledge

The first task performed by the ADSS (see Figure 18) is the **Data Collection**, which consists of the creation and initialization of the AKB with application-specific information. The ADSS performs, by means of compiler tools, an analysis of the application source code to identify the assets and all relevant information about the application (like metrics, profile information, code size, slices, points-to information, etc). The information extracted with this activity will be named *application-specific knowledge*. Additionally, other information, including user preferences and constraints, is

[6]The AKB is formed by several abstract models (one main model and several sub-models) which represent all the information that is needed to the ADSS to perform its job. These models have been introduced in D4.01 and will be continuously updated during the project lifetime. These abstract models formally describe the context of MATE attacks by precisely representing the relationships between assets, threats, attacks, protections, and attack attributes, as we informally described in D1.02. In the same deliverable, this information has been categorized as a priori knowledge and execution-specific knowledge. The a priori knowledge is valid for all applications of the ADSS and captures all information that is generally applicable and relevant in the context of software protection against MATE attacks. The a priori knowledge permits the identification of what kind of protections can be theoretically used depending on the attacks and assets, their dependencies, how they could be used in combinations, etc. Additionally, the a priori knowledge allows an initial approximate estimation of the impact of protections the applications that will be refined when the application specific data will be added.

gathered from the user via a user interface. All the knowledge acquired during the data collection phase is added to the AKB, however, this information can be optionally presented to the user before actually being added to the AKB, that is, the ADSS will present to the user a (*validation report*) including a set of facts that need an approval by the user before being inserted in the AKB.

13.1.2 Combination Generation: identifying the suitable combinations

At the end of the Data Collection phase, the application-specific knowledge is formed and it is ready to be used by other modules for the second phase in Figure 18, the **Combination Generation**, which has as target the generation of the suitable combinations forming the solution space. To generate the suitable combinations, the a priori knowledge together with the application-specific knowledge needs to be extended to the execution-specific knowledge[7].

The generation of execution-specific knowledge is performed by means of an **Enrichment Framework**[8], a component of the ADSS used in the Combination Generation phase that will reason about the a priori knowledge and the application-specific data to derive new knowledge using several techniques. After the enrichment is finished, the execution-specific knowledge can be queried to determine the suitable combinations, that is, an exhaustive search of all the suitable combinations possible. However, the exhaustive search is not our objective. The suitable combinations that will be in the solution space are not all the possible combinations that can be deployed on the target application. Depending on the security requirements and the user preferences, only the most appropriate suitable protections will be inserted in the solution space. When the execution-specific knowledge will contain dominance relations (for instance, all the combinations using the protection X are better in all possible evaluation fields than combinations including protection Y) or other associations that permit the ordering of protections or combinations, these properties will be used to insert only the meaningful combinations in the solution space.

The general architecture of the ADSS Enrichment Framework used in this phase, sketched in Figure 19, includes an Enrichment Coordinator that manages a set of Enrichment Modules (EMs), each one able to perform a specific reasoning task based on the information in the AKB. Figure 19 shows that the AKB is directly connected to an Ontology reasoner. This follows from our decision that the AKB will be a formal ontology represented in OWL-DL.

Through the enrichment process, the ADSS is made able to infer new information by the means of a completely customizable logic system that overcomes the limitations of ontological systems and offers a very powerful and flexible approach. For instance, EMs can use different formal methods:

- *Ontology-based tools* use ontology fragments and ontology-based reasoners to optimize enrichment of specific concepts. It is often convenient to separate fragments of the main ontology to perform ad hoc optimized reasonings in independent knowledge domains. These EMs are presented in section 15.3.
- *External non-ontological reasoning tools* use other semantic reasoners. In general, these tools are able to infer logical consequences starting from a set of axioms and a set of inference

[7]As explained in D4.01, the execution-specific knowledge is obtained by customizing the a priori knowledge on the target application based on the application-specific knowledge. The execution-specific knowledge permits the determination of which are the protections that can be actually deployed. Indeed, based on the application-specific information collected during the source code analysis, it is possible to identify the attacks that can be effectively mounted against the assets, determine the protections that are effective against these attacks and that can be actually deployed given the source code properties, the attacks that can possibly render unusable the deployed protections, the protections against these new attacks, and so on. This process will eventually produce all the possible combinations of protections that strengthen each other and secure the application assets, that is, the suitable combinations. Additionally, the execution-specific knowledge gives a more precise estimation of the impact of protections on the application.

[8]the term enrichment is borrowed from the ontology world, where enrichment is the name given to the process that infers new axioms (that is, new knowledge), from the axioms already available in the knowledge base by means of inference rules.

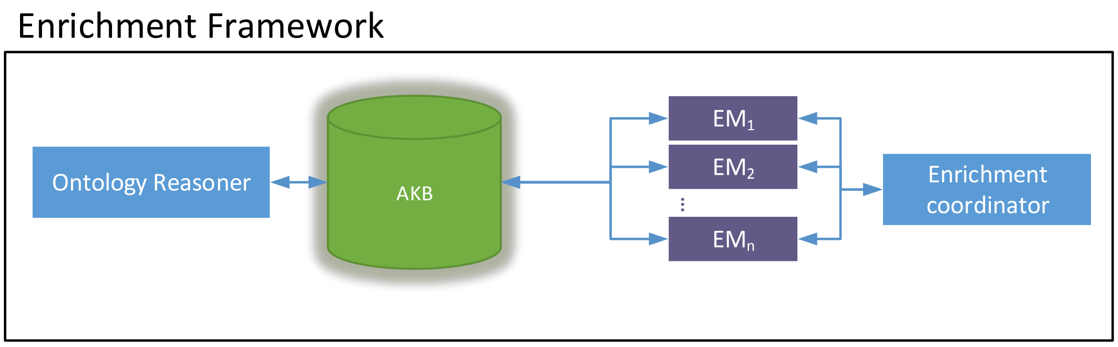

Figure 19: A sketch of the Enrichment Framework.

rules (e.g., tools able to perform backward/forward chaining). They are usually founded on some well known and coherent logical system, like First Order Logic (FOL). They can be used instead of ontology methods because they can perform reasoning that is not feasible with a DL ontology (because they use a more expressive logic system) or because they can compute the same results as the ontology but in a much more efficient way. These EMs are presented in section 15.2.

- ASPIRE *tool flow components* will be able to analyse the application and autonomously determine several protection profiles and add them into the AKB. A protection profile is defined as an association between one protection tool component and a set of its configuration parameters which univocally determines one of the possible ways to deploy a protection with that tool.

The Enrichment Framework can also be successfully applied when the human intervention is needed, for instance, by asking the user for an information item missing in the knowledge base via a user interface. Interactive enrichment modules are also known as *user interaction modules*, since they require a manual intervention.

In ASPIRE, the tools on Diablo form an important example of the tool flow components enrichment. These tools will be composed of several modules each one implementing a protection technique or step thereof. Some of those modules, like the binary obfuscation module as it pre-existed ASPIRE for the protection of x86 code, have been developed (in the pre-ASPIRE era) with an internal optimization logic that analyses the application's binaries to determine the characteristics that could affect the protection deployment. The results of the analysis of the internal optimization logic is then used to identify several efficient ways to deploy the protection modules implement. When Diablo is used as a stand-alone tool, each invoked component selects the best solution: each Diablo module performs a local optimization as it selects the best way to deploy one protection.

In the ASPIRE context, each Diablo module will be able to insert the set of identified profiles into the AKB. Moreover, the modules will also provide an evaluation of the impact of each protection profile, that is, it is also able to estimate the effects of the implementation of each profile and to rate its effectiveness.

When Diablo is used inside the ASPIRE context, each Diablo-produced profile is evaluated by the ADSS together with all the applicable techniques when performing the global optimization to select the golden combination.

13.1.3 Evalution: associating measurable features to suitable combinations

After all the suitable combinations are inferred, the **Evaluation** phase is performed (see Figure 18), which has the objective to associate suitable combinations in the solution space to measurable features. Also in this evaluation phase, the results will be written in the AKB.

Before this phase, the following is available in the AKB for evaluating suitable combinations:

- metrics computed on the original application, which quantify the measurable features on the application;
- application-specific knowledge, which describe all the relevant characteristics of the application that can be used to determine vulnerability to attacks and suitability of protections;
- protection profiles, which estimate the impact and the effects of the implementation of single protections added by ACTC components in the AKB;
- enrichment data, qualitative relations which are part of the execution-specific knowledge obtained by the Enrichment Framework.

These data are not enough to evaluate suitable combinations for the purpose of selecting the best one by means of an optimization program. Therefore other techniques must be used to quantify suitable protections to be maximized or minimized by the target function.

Therefore, this phase will see the extensive use of the ASMs, based on Petri Nets, which have the objective to determine if an attack is feasible against the target application and to simulate the impact of protections on each attack step. Additionally, ASMs – already introduced in D4.01 – will be used to estimate impact on protection performance and other effects on the application. Effects of protections and combinations will be estimated by simulating changes into the ASMs. The changes to impose to ASMs will be determined by the relations available in the AKB.

ASMs are one instance of the *composition models*, needed to evaluate the suitable combinations. Composition models will produce mathematical formulas, lead by associations between protections and attacks in the AKB, that will be taken as input when generating the optimization program. Composition models are under investigation and are still an open research issue that will be addressed during the ASPIRE project.

One possible approach is to use the metrics framework (that will be initially documented at M12 in D4.02), whose primary goal is to capture the main application characteristic[9], to formulate observations in terms of the impact of protections and impact of attacks, because of the link that protections and attacks have with measurable factors. However, we will investigate the possibility to define (predictive) composition models, built on on data obtained by the metrics framework on test applications. Composition models will allow us to estimate how the deployment of a protection (invoked with a set of parameters) or a combination of protections will impact different metrics. These models might determine the impact by invoking tools that apply the transformation and measure the result (e.g., for EMs based on ACTC components), but they might also do it by estimating the impact. That is, instead of a posteriori verifying the impact on protection by actually computing the metrics, we will try, by means of predictive models, to give a good approximate a priori estimation of the impact of combinations of protections without actually deploying the protection. Estimations rather then correct computations are needed because (1) not all combinations can be tried and measured, hence definitely the effects of some combinations will be estimated from the effects (measured or estimated) individually for the protections being

[9]ASPIRE aims at addressing the evaluation problem by measuring the impact of protections and combinations of protections on the target applications before and after the deployment of the protections. The types of comparisons include, but they are not limited to, comparing applications using several metrics (diff, size, change impact), or comparing performance, speed, memory and deriving execution overhead.

combined, (2) estimating effects may be much faster than computing it exactly even for individual protections of which the analysis tools present impact in terms of metrics to the ADSS.

These composition models can also be based on probability theory. Attackers can use sound and unsound attack steps, all of which have different levels of precision and completeness. In many cases, there is no guarantee that some attack steps or method will succeed or fail given some protection, simply because either reasoning is limited or model accuracy is also insufficient. However, in these cases, it is therefore possible to determine the likelihood in terms of probability of success.

13.1.4 Pruning: reducing the solution space size

Given the large number of available protection techniques and the different ways they can be used to protect application assets, it can become necessary to provide a number of strategies that limit the solution space. As anticipated before, an optional phase, named **Pruning** (see Figure 18), can be executed to discard the combinations that are less likely to be selected by the solver.

Combinations are not discarded based on the value of the target function that will be used in the optimization program, rather they are discarded on individual measurable features. For instance, a user may decide to exclude all the combinations that add a performance overhead more than 10%.

As pruning works in a greedy way, it can lead to a suboptimal golden combination. We only accept this compromise when the solution space is estimated too large to be solved in a reasonable amount of time. At the end of the process, the worst suitable combinations (according to the individual measurable features) will be removed from the solution space, hence ignored during the optimization phase.

The most suitable approach for pruning is to adopt a set of user-specified criteria, such as removing the combinations whose value for a measurable feature is below or above a given threshold. Therefore, this step may require some additional manual input from the user to configure the pruning parameters.[10]

Apart from efficiency and security reasons, the pruning of the solution space can be also useful to exclude some unwanted protection combinations a priori. For instance, the ADSS user can choose to discard all the combinations that can be deployed with tools that would cost him too much, in other words, the user can constrain the tool flow to use only free protection tools. A set of predefined discard rules will be developed by the ASPIRE consortium and made available for user selection in the Pruning phase.

The best point in the workflow to perform pruning is just after the Evaluation phase, where also some ASM results or composition model data can be used for some more sophisticated pruning. However, we are considering other approaches to reduce the solution space size. Pruning can be done during the generation of the combinations, in this case it will be named *early pruning*. For example, if some combination is using a technique whose performance is known in advance as poor (e.g., because of protection profile information added by ACTC components), or some technique is using protections which are known in advance to work together inefficiently, the generation process can be accelerated by constraining the enrichment and avoiding the generation of these combinations.

Additionally, for performance reasons, we are also investigating the possibility to perform *late pruning* by adding constraints (e.g., inequalities) during the Optimization phase. That is, we will evaluate during the ADSS implementation two options:

- to filter suitable combinations before starting the Optimization phase, to generate and solve

[10]Pruning criterion can be also provided by the user during the Data collection phase.

a smaller program (standard pruning), or

- to skip the Pruning phase and generate a larger optimization program that includes all identified suitable combination then filtering the solution space by means of a set of constraints (late pruning).

13.1.5 Optimization: computing the golden combination

The **Optimization** phase (see Figure 18) solves the ASPIRE decision problem for the target application. The optimization can be split in four steps, as shown in Figure 20:

1. The user chooses what he wants to optimize by selecting an *optimization profile*.
2. Based on the selected optimization profile and the (non-pruned) suitable combinations already in the AKB, an optimization program is prepared by a profile-specific Optimization Program Builder module.
3. The solver takes as input the optimization program and determines the golden combination.
4. The golden combination is written into the AKB.

The use of optimization techniques requires a formal description of what is the best combination of protections for the application assets, that is, to characterize how to select the golden combination. Nevertheless, abstractly defining the golden combination using solver specific commands is in general outside the possibility (or the taste) of several software developers. For this reason, there will be several *optimization profiles* available for the user to choose from to build the optimization program for them. Optimization profiles on one hand describe to the user what will be optimized by the ADSS using simple words, i.e., they describe the target function. On the other hand, an optimization profile formally represents all the necessary technicality that allows its use within the ADSS to find the actual solution of the ASPIRE decision problem it entails.

A profile thus points to two items:

- *The solver* to be actually used to determine the golden combination. It is known from literature that there are solvers that are able to find the best solution (or a reasonably close approximation) faster than others. There is not a single winner, however, and the most appropriate solver to use depends on the problem degree, the type of variables, or the peculiarities of the problem to solve. See `http://plato.asu.edu/bench.html` for an up to date resource on solvers benchmarks.

  Optimization programs are classified according to the degree of equations and inequalities used: there are linear, quadratic or polynomial problems. Additionally, optimization programs can be classified according to the types of variables used by equations and inequalities, there are Boolean, integer number or real number valued problems. Most of the solvers only solve linear programs, while, only a few solvers can treat quadratic or polynomial problems. Moreover, not all the linear solvers also solve integer linear programs and Boolean programs often have dedicated tools. Therefore, we preferred not to choose one solver for the ADSS, we associate the solver to the optimization profile. The time to solve an optimization problem dramatically increases depending on the problem degree. The general aim is to minimize the degree of the polynomial problems. Consequently, we will build *Linear programs* (LP) or *Integer Linear Programs* (ILP). Further information on the subject is available in Appendix Section E.2. LP and ILP can be applied to a great number of practical interest fields. Industries that use this kind of models include transportation, energy, telecommunications and manufacturing. Our preliminary analysis and our experience in previous projects assures that they can address most, if not all, the optimization problems we need to solve in ASPIRE. The need for higher degree problems will be evaluated during the project lifetime.

- *the Optimization Profile Manager*, a software component[11], which is able to read the suitable solutions that have not been pruned and their evaluation, generate an optimization program, and pass it to the solver indicated by the profile (steps 2-4 in Figure 20). That is, the optimization program builder is able to produce a file that follows the solver-specific syntax or it is able to call the solver-specific API to pass all the optimization program instructions;

Together with the optimization profile that will be developed during the ASPIRE project, users or other stakeholders in the potential ASPIRE marketplace are allowed to customize existing optimization profiles or define new ones.

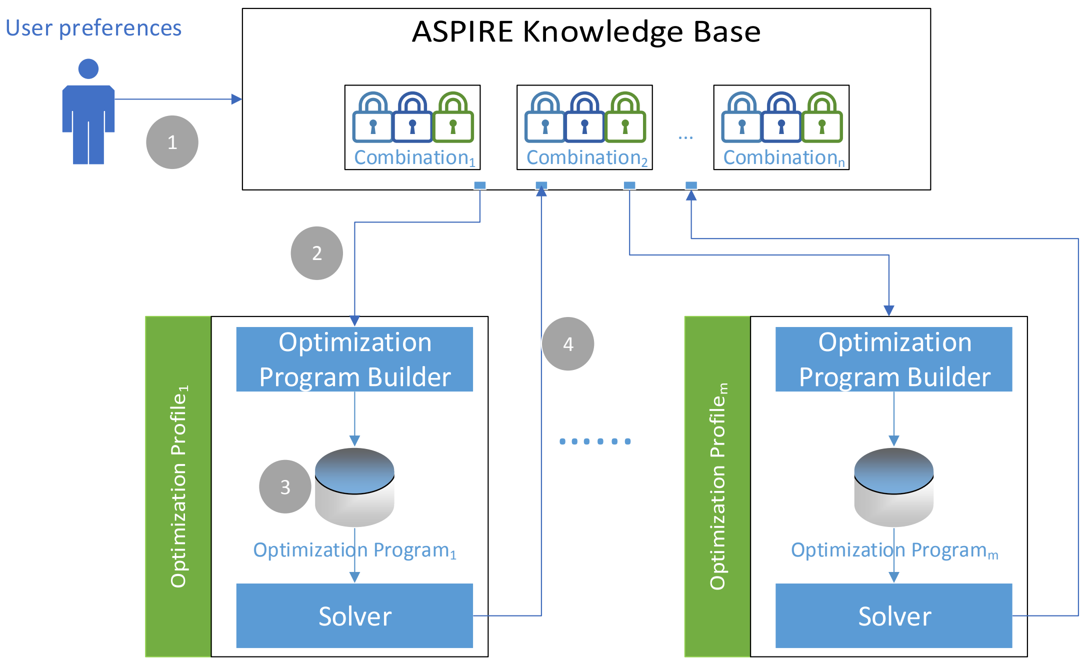

Figure 20: Schema for the optimal protection combination detection.

13.1.6 ACTC Instruction: communicating the ACTC how to enforce the golden combination

Once the golden combination has been computed, the ADSS work is almost finished. The only remaining task is to actually deploy the protections in the golden combination. Therefore, the **ACTC Instruction** phase is performed (see Figure 18), which produces as output the instructions. for the ASPIRE ACTC. The ADSS communicates to the ACTC the protections to use, and for each protection the options, flags and protection components to enable.

The output will consists of two parts: (1) instructions that select the protections to use and, for each protection to use, communicate the protection options and flags to use (e.g., some scripts to execute each of the protections), and (2) annotations, that are added to the application source code to mark the parts that need to be protected and refer to the protections to use. In particular, security requirement annotations (present in the application source code passed to the ADSS) are

[11]theoretically, it is needed one Optimization Profile Manager for each profile but we aim at reusing some of them within the ASPIRE scope. For the future, we expect that the developer of the optimization profile also produces this software component.

expanded into protection-specific annotations to be processed by the ACTC. Probably the most effective way to configure protections is by adopting code annotations, i.e., protection-specific code annotations and protection profiles. Some of these annotations will be the modifications of manually written security requirements, others will be added by the ADSS. This approach is certainly elegant and allows a complete separation of ADSS and ACTC, which will be very useful during the project development. There is some redundancy in reading and writing annotations from the source code, however, that can be avoided. Therefore, we are also investigating the possibility to directly produce facts that are usable by the ACTC.

13.1.7 Backtracking

As anticipated, some of ACTC components can fail during the actual protection deployment. The main reason is that the approximations made in some of the models in the AKB (application details, protection restrictions, other requirements) might not always allow to anticipate the deployment issue.

Our goal is to continuously enrich the AKB so that these failures will disappear. However, since this is practically impossible given the complexity of the protection deployment task, backtracking will still need to be performed (see Section 1).

Practically, backtracking consists of being able to parse, understand, and insert in the AKB the output of the tools, which will include error codes, line numbers, and other information useful to locate and solve the problem.

During the project lifetime, we will not support automatic backtracking, mainly because it is more an industrialization issue than a research problem. Therefore, users will be asked to manually add the relevant information that they were able to extract from the failed protection deployment. To ease the input of the failure information in the AKB, we are developing an Eclipse Rich Client Platform (RCP) application based on XText (`http://www.eclipse.org/Xtext/`) that takes as input the AKB ontology and allows users to add valid individuals, object and data properties with support of autocompletion, and automatic validation.

13.2 Architecture overview

The ADSS consists of multiple specialized modules as sketched in Figure 21. The complete specification of the ADSS architecture is presented in Appendix H.

The central repository of the ADSS is the *AKB*. Within the ASPIRE project we will develop it as an ontology and represented in OWL. As stated before, the AKB contains all knowledge that is necessary for all ADSS components to perform their operations and the output their results.

The principal software components of the ADSS are:

Data Collector, which performs the Data Collection phase by means of several analyzers and GUIs.

Protection Combination Generator, which performs the Protection combination phase. It contains the Enrichment Framework.

ADSS Pruning Module, which performs the Pruning phase;

ADSS Optimization Module, which performs the Optimization phase. It contains an internal module that performs the Evaluation phase, only for the combinations in the solution space and for the measurable features required by the optimization profile;

ACTC Bridge Module, which instructs the ACTC on how to deploy the golden combination.

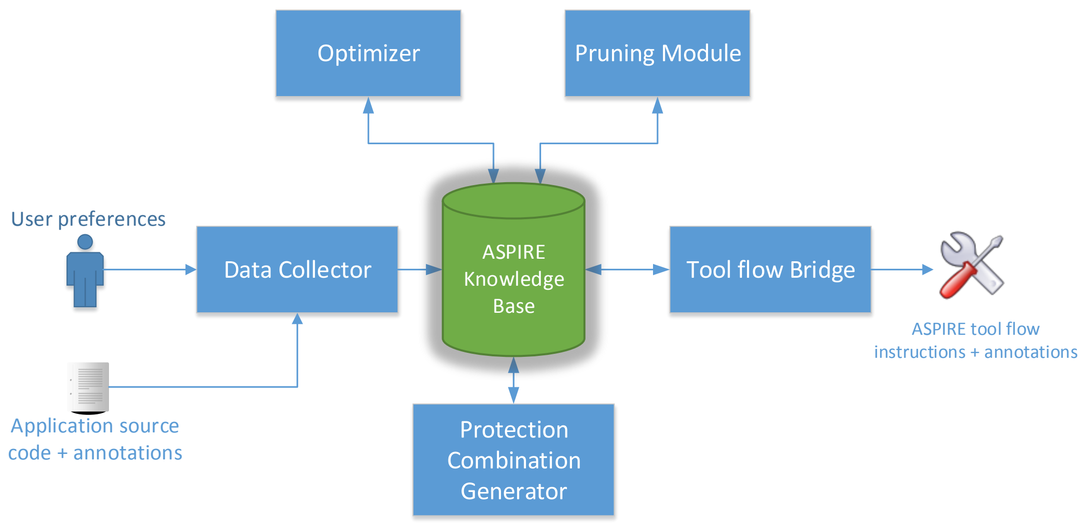
Optimizer
Pruning Module
User preferences
Data Collector
ASPIRE Knowledge Base
Tool flow Bridge
ASPIRE tool flow instructions + annotations
Application source code + annotations
Protection Combination Generator

Figure 21: The ADSS architecture.

13.3 Implementation

The ADSS developed during the ASPIRE project will be a Java Eclipse Rich Client Platform (RCP) application.

The Eclipse platform is designed an open and extensible platform so that its components can be used to build any client application. An RCP application is a layered software architecture that provides a set of basic services (including a runtime environment, application model, dependency management, rendering engine), and a workbench. This software architecture can be extended by adding several plug-ins. Plug-ins are connected through so called extension points.

Therefore, all the ADSS components will be developed as Java Eclipse plug-ins.

The advantage of an RCP application is that we can build the ADSS on well-known and appreciated computing platforms instead of having to write a complete application from scratch. Building on a platform facilitates faster application development and integration, while the cross-platform burden is taken on by the platform developers. The platform allows the seamless integration of independent software modules like graphic tools, spreadsheets and mapping technologies into a software application with a simple click of the mouse. Their creators claim that programs built with RCP platforms are portable to many operating systems while being as rich as client-server applications that use so-called fat clients or traditional clients. This will certainly reduce the development time, a real advantage for a three year research project. Additionally, the result will be an application whose look and feel, and functionality are familiar to many developers. The only negative aspect of the use RCP applications, for our purposes, is that they are usually large (as they have to include the common parts of the Eclipse platform) and they perform slightly worse than ad hoc application (that, on the other hand, require years to be developed at the same level of usability).

All the components will be implemented as a set of Eclipse plug-ins using the Java programming language. Each component must declare an interface in order to be correctly 'registered' in the ADSS. This interface is the API defined in the previous section and will be specified using the extension point/extension mechanism in the Eclipse framework[12].

For ontology reasoners and mathematical solvers, we have already investigated several products.

[12] See `http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points`.

The candidate ontology reasoners include:

- Pellet (`http://clarkparsia.com/pellet/`), a powerful and highly-customizable reasoner that includes multiple inference computation algorithms. It is our first candidate to become the main ontology reasoner.
- HermiT (`http://www.hermit-reasoner.com/`), a reasoner based on hyper-tableau calculus, also suitable for large scale ontologies.
- FaCT++ (`http://owl.man.ac.uk/factplusplus/`), a fast and lightweight reasoner written in C++, very fast on small and medium sized ontologies.

There are several other reasoners we have considered, a full list is available at `http://www.w3.org/2001/sw/wiki/OWL/Implementations`.

The candidate solvers include:

- lpsolve (`http://lpsolve.sourceforge.net/5.5/`), a free mixed integer linear programming solver based on the revised simplex and the branch-and-bound methods. It is our first candidate to become the main solver for all the LP problems.
- CPLEX (`http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud/`), a commercial solver by IBM, designed to work with large-scale mathematical linear and quadratic optimization problems. It shows the best performance but since it is free only for academia, only academic partners will use it for internal experiments. We will consider its use only if we'll need to solve quadratic problems.
- MOEA (`http://www.moeaframework.org/`), is a free and open source Java library for solving multi-objective programs that supports supports genetic algorithms, differential evolution, grammatical evolution, etc. It is our first candidate to become the main solver for all the multi-objective problems.

A comprehensive list of solvers is available at `http://en.wikipedia.org/wiki/List_of_optimization_software`.

Finally, POLITO has developed a Java Ontology API, which provides very powerful CRUD (Create, Read, Update, Delete) interface to OWL ontologies which will be used to access AKB data. The Java Ontology API is already available to the project consortium. It is currently under investigation the possibility to publish it through a portal for Open Source Software.

14 Enrichment examples

Section authors:
Cataldo Basile, Daniele Canavese

This section presents a set of examples that explain our approach to the enrichment, i.e., the deduction of execution-specific information from the a priori information and collected application specific data. We will present two examples, which describe two different types of deductions and will be the basis for the development of two Enrichment Modules. The first example shows an inference engine based on backward programming (and implemented in Prolog) to deduce attack paths that can be mounted against application assets. The second example shows how an ontology-based deduction system can be used to deduce when a given protection technique can be used to mitigate attacks against application assets. In particular, the case will focus on anti-debugging and code flattening protections.

14.1 Deducing attack path

This section presents an example of the enrichment method we want to use to automatically infer the attack paths that may be executed to mount attacks against the assets annotated in the target application. The purpose of this example is to present an insight on the modelling effort done in T5.2, which will produce a set of independent inference models that will be integrated in the Enrichment Framework. The selected example has been built on a non-ontological logic system, i.e., prolog, and this further justifies the need for the ASPIRE Enrichment Framework. Additionally, we present our advancements in the modelling approach, as we document the way we automatically build attack paths against applications on the security model presented in D4.01 (Section 3) and how the AKB is used to this purpose, and in general, for enrichment purposes. Finally, this section is also useful to show the advances in modelling within the ASPIRE project: we started from an informal description of attacks (D1.04), we manually generated the associated Petri Net to prove the validity of simulation models (D4.01), now we are building an automatic inference system to deduce attack paths so that the simulation models can be automatically obtained and used for the Evaluation phase.

In literature, a similar inference engine has been developed in a different context: the identification of attacks in network security based on traditional risk analysis [3, 4]. Some more bibliography was already presented in D4.01.

We will present this enrichment example by starting from the Informal Vanilla OTP Attack Paths Description (from D1.02, Section 5.3.2). The ultimate objective of the attack is being able to impersonate the application user, therefore the attacker needs to be able to generate all the one time passwords (OTPs). This can be achieved by obtaining the seed that is used to generate all the OTPs. The seed is obtained from the application server and is the unique identifier of the application. However, to correctly generate the OTPs the attacker also needs to identify the initial value of a counter, which is updated[13] after a new OTP generation and whose initial value is hardcoded in the application. Finally, to use the OTP generation function in the application, the user needs to know and provide the PIN to authenticate him before generating the current OTP.

The Petri Net associated to the Informal Vanilla OTP Attack Paths presented in D4.01(Section 2.3,

[13]Even if it is named "counter" in this example, the attacker does not know in advance whether this variable will be incremented by one. Counter is the name given in literature to the additional input passed to the OTP generation functions, which is needed to generate every time a different OTP. The value of this variable can be incremented by different step values, or a pseudo-random number generator is invoked at every generation, or any other function shared between the OTP generator and verifier.

Figure 8)[14] is a fruitful exercise that successfully proved that attack paths can be formally modelled by means of Petri Nets. However, it required human expertise (already informally coded in D1.02) to determine all the possible strategies to mount the attack, and manual effort to transform the sequences of informally described operations into an optimized and semantically equivalent Petri Net. The effort required to model this single attack (although it has an advantageous learning curve) is not scalable enough to be actually used by the ADSS. Indeed, it is not possible to imagine a scalable decision support system (that must target any kind of applications) where all the attack paths are all manually added by experts.

Therefore, we propose to develop a method to deduce the attack paths automatically. The definition of this deduction method changes the focus from the definition of an entire attack path to formal modelling of single attack steps. However, this approach requires the creation of a logical system that is able to compose single attack steps into attack paths that permit the attacker to achieve its goals. We will show in this section that, if the attack steps are carefully modelled, a powerful form of enrichment based on backward reasoning (which has been already sketched in D4.01, Section 3.5) can be used to automatically infer new attack paths.

It is worth noting that the automatic deduction is not in conflict with manual description of attack paths. Attack paths that have been already described or identified, e.g., by users that prefer to insert new attack paths manually, are supported by our approach, as the AKB is able to store attack paths and Petri nets associated to them, as shown in D4.01.

14.1.1 Modelling the attack steps

Before describing how a goal-oriented approach (backward programming) can be implemented to automatically discover attack paths, we will re-interpret the Vanilla OTP Attack Paths. We will assume that, for backward programming purposes, there will be a *fact base* in place, where a set of fact. *Facts* are data available to the reasoning. For instance, facts may be assertions that state that some parts of the application to protect are assets, or assertions that report that some attack steps have been already performed or some preliminary sub-goal to mount the attack has been achieved. Several data from the AKB are actually facts needed for the reasoning and can be imported into the fact base, e.g., facts about the applications, protections, and the available attack steps. We anticipated in the previous section that, for an attacker that wants to mount an attack on the vanilla OTP generator, the final goal of the attacker is to be able to use the OTP generation function every time he wants to impersonate a user. To achieve this goal he needs to:

- bypass the PIN-based authentication phase, needed to use the generation function;
- know the counter;
- know the seed.

These achievements can be considered as the three sub-goals in which the attack goal is split. In the attack sub-model presented in D4.01, we introduced the association `decomposedAND` to formally model that the sub-goals are AND-ed. Here, we will therefore present how to model the attack paths to achieve these three subgoals.

First sub-goal: obtaining or bypassing the PIN

To identify and defeat an authentication procedure based on something the user knows like the PIN-based authentication scheme that protects the OTP generators, that is, the **first sub-goal**, an

[14]There are two major differences between the attack paths identified in this example and the attack paths determined by the Petri Net in D4.01. The Petri Net in D4.01 only considers two sub-goals, obtaining the seed and bypassing the PIN, and these two sub-goals are serialized instead of being AND-ed.

attacker needs to (1) steal the secret, i.e., the PIN, or (2) locate and bypass the authentication function code.

To steal the PIN, an attacker may inject in the application some code, running on the victim's device, that sends to a server maintained by the attacker the PIN (or use other social engineering approaches that are outside the scope of this deliverable).

To simplify the example, we will omit in all the attack steps an important pre-condition: the attacker must own the proper tools to perform the analysis (static analysers, dynamic analysers, and other attack tools that have been already listed in D1.02) and it must have the expertise to use them. These pre-conditions will be considered in the enrichment framework. The protection requirements sub-model in D4.01 uses the class `ProtectionTarget` and all the associations from `Protectiontarget` instances to restrict the tools an attacker may use, and the association `hasExpertise` to bind an `Attacker` instance to an intance of the `AttackerIdentificationEnum` class.

The following attack steps concur to achieve this goal:

1. *Locate the PIN-based authentication function code*. This step has no pre-conditions, and produces as output the knowledge about the code area where the authentication function is located.
2. *(Statically) skip a code fragment*. This step has as pre-conditions the fact that the code area where the code fragment to skip is located, is known and produces as output a fact: "the PIN-based authentication function code fragment can be skipped".
3. *(Dynamically) skip a code fragment*. This step has as pre-conditions the fact that the code area where the code fragment to skip is known and produces as output a fact: "the PIN-based authentication function code fragment can be skipped".
4. *Set up a server where to store PINs*. This has no pre-conditions and produces as output two facts: "the server to store PINs is available" and "the protocol to communicate with the server is known".
5. *Steal the PIN by injecting code* that sends to a server maintained by the attacker the PIN. This has as pre-conditions the fact that the code area where the code fragment to skip is located is known, the server to store PINs is available to the attacker and the protocol to communicate with the server is known. It produces as output a fact: "the PIN is known".

After having presented the individual attack steps, we provide hints on how the backward chaining works to compute the attack paths. This first sub-goal is achieved when in a fact base there is the fact "the PIN is known" or the fact "the PIN-based authentication function code fragment can be skipped". Therefore, the inference engine looks for the attack steps that produce as output the facts, and discovers the attack steps 2, 3, and 5. Step 5 has as pre-condition the fact that a server to store PINs is available (thus step 4 needs to executed before step 5) and that the authentication function is located (thus step 1 needs to be executed before step 5). Analogously, it is possible to detect that step 2 and 3 both require the location of the authentication function produced by the step 1. Finally, the attacker needs to have the expertise to use the attack tools that serve to mount the attacks, as said before. Thus backward chaining would detect the following attack paths, composed of ordered steps: step 1 then step 2, or step 1 then step 3, or step 1 AND step 4 (can be executed in parallel) then step 5.

Second sub-goal: obtaining the counter value and update procedure

To obtain the counter value, the **second sub-goal**, the attacker has to observe the OTP generation phase, to read the value of the counter when it is used and to determine the counter update function. Once he has this information, the attacker can autonomously update it.

The following attack steps are needed to achieve this sub-goal:

1. *Locate the OTP generation function code using dynamic analysis*. The OTP generation function uses the counter[15] to generate a new OTP. This attack step has no pre-conditions and produces as output the knowledge about the code area where the OTP generation function is located.

2. *Perform static analysis of the OTP generation function code area* to identify a well known pattern, the chain of XOR operations that are performed during the OTP generation. This attack step has as pre-conditions the fact that the code area where the OTP function is located is known, and produces as output a fact: "the code where the counter is used is known" and "the counter update procedure is known".

3. *Perform dynamic analysis of the identified OTP generation function* to determine the counter value. This attack step has as pre-conditions the fact that the OTP function code is known, and produces as output a fact: "the counter value is known".

The second sub-goal is thus achieved when the facts "the counter value is known" and "the counter update procedure is known" are in the fact base.

Third sub-goal: obtaining the seed value

To obtain the seed value, the **third sub-goal**, two attack methodologies can be identified. First, the attacker can observe the OTP generation to get the seed when it is used, or, second, it has to intercept the seed when it is sent by the server to the application during the seed provisioning phase. It is worth recalling that the provisioning phase consists in the decryption of the some data received from the server that contain the seed.

The first methodology is the analogous of the identification of the counter value. More precisely, the attacker needs to identify the same function, the OTP generation function, where both the seed and the counter are used and produce as output the fact "the seed value is known" by performing the same attack steps as before[16]. The analogy between the identification of the counter and the seed suggests a possible improvement of this simplified inference system: some attacks and attack steps may become *parametric attack steps*. For instance, the aim is to define an attack step named *statically locate function* which takes as input the name of the function to locate (which must be annotated in the source code) and produces as output the fact that the code area is located. Analogously there will be the *dynamically locate function* attack step.

To achieve the third sub-goal with the second methodology, which involves stealing the seed during provisioning, the attacker has to deal with application-specific characteristics. In this case, the provisioning phase is performed only once for each client when it first connects after the application is installed. Therefore, on an untampered application, the attacker cannot execute more than once the provisioning phase if he does not reinstall the application. Installing the same application again and again is a suspicious activity that application developers could monitor. To avoid this provisioning phase limitation, the attacker can build a fake server which substitutes the original application server when performing the provisioning phase. Of course, this operation requires the observation and understanding of the provisioning protocol messages (or some reply attack if possible). If the fake server is available, the attacker can also tamper with the application to remove the limitation on the execution of the provisioning phase that oblige to perform this phase only once (in a very similar way to the one that serves to bypass the PIN limitation code).

[15]Note that the same function also uses the seed thus this phase can be shared with the other sub-goal.

[16]From the modelling perspective, the delicate task we are performing in tasks T4.1 and T5.2 of the project, is to abstract from this single application-specific characteristics and to derive a model tool that can also be used for other applications.

To abstract from the above application specific characteristics, we introduced in our model a variable that is used to store the number of execution of a given function, the `execution counter`, which only serves from the modelling perspective[17]. Analogously, we need to count the number of executions of an attack step in attack paths and to put a restriction on this value. For the same abstraction purposes, we also have to model the fact that an attacker has to execute a function more than once to be able to steal data or understand functions. That is, our model permits the association of variables to facts, e.g., variables that characterize attack steps and/or application-specific information. The use of variables makes our model very expressive. As a first consequence, the attack steps and attack paths identified in this section are extensible to all applications that include as assets functions that can be executed only once, or have assets (e.g., data/variables) in functions that can be executed only once.

The second attack methodology can be achieved with two different attack paths. The first one requires the use of the following attack steps:

1. *Re-install the application*. This attack step has no preconditions and puts to zero the counter of the number of executions of the provisioning function.
2. *Dynamically analyse the un-tampered application when it interacts with the (original) application server to locate the seed decryption function*. This attack step has as pre-condition the fact that the number of executions of the provisioning function is less than one. The effect of this attack step is the increment of the `execution counter` and the increment of the counter of the number of executions of this attack step.
3. *Dynamically analyse the un-tampered application when it interacts with the (original) application server to read the seed* when it is output by the decryption function. This attack step has as pre-condition the fact that the attacker knows where the seed decryption function is located. It produces as output the fact "the seed value is known".

Alternatively, the second attack approach can be also mounted with another attack path, which first skips the provisioning limitation code by tampering the application and developing the fake servers. This attack path first requires the following attack steps:

1. *Locate the provisioning limitation code*. This step has no pre-conditions, and produces as output the knowledge about the code area where the fact that the provisioning has been executed and tested is located.
2. *(Statically) skip a code fragment*. This step has as pre-conditions the fact that the code area is known where the code fragment to skip is located, and produces as output a fact: "for the tampered application the provisioning limitation can be skipped".
3. (Dynamically) skip a code fragment. This step has as pre-conditions the fact that the code area where the code fragment to skip is known and produces as output two facts: "the OTP generation function in the application is tampered" and "for the tampered application the provisioning limitation can be skipped".

However, since the (original) application server may notice that the provisioning is executed more than once from the same application, he also has to execute the following attack steps:

1. *Develop a fake server* able to provision a valid seed to a client whenever is requested by the application. This attack step has no pre-conditions but the fact that the attacker has expertise

[17]The application source code will probably contain a variable with the same purposes, however, the execution counter is only a modelling abstraction and, to achieve his goal, an attacker must not necessarily look for it into the application code.

to build this kind of server, and produces as output the facts "there is a tampered seed provisioning server" and "the tampered server does not notice that the provisioning phase is executed more than once".

2. *Dynamically analyse the tampered application* when it interacts with the tampered application server to locate the seed decryption function. This has as pre-condition the fact that the attacker owns and has expertise to use some dynamic analysis tools, there is a tampered application, there is a tampered server, for the tampered application the provisioning limitation can be skipped, and the tampered server does not notice that the provisioning phase is executed more than once. It produces as output the knowledge of the seed decryption function.

Finally, the attacker has to execute the attack step *dynamically analyse the un-tampered application when it interacts with the (original) application server to read the seed*. This has already been presented before and produces as output the fact "the seed value is known".

The third-sub goal is thus achieved when the fact "the seed value is known" is in the fact base.

To summarize the modelling effort presented in this section, an attack step is defined by the following parameters:

- preconditions determine when an attack step can be actually executed. Preconditions are predicates on the fact base, the database of all the known facts[18].
- output, i.e., a list of facts that are added to the fact base.

Although it is not needed for the formal model of the enrichment method to automatically infer the attack paths, we will associate each attack step to a textual description to be read by humans (e.g., when the discovered attack paths are reported).

14.1.2 Definition of the inference

After having defined the single attack steps, it is necessary to define the logic to build the attack paths. This can be easily defined using backward reasoning and an inference engine. We actually implemented this example in Prolog. Omitting the pre-conditions on expertise and tool availability, we present a simple example of how to model this example. First of all, each attack step is defined with two functions:

```
attackStep(Identifier, ListOfProducedFacts)
attackStepRequirement(Identifier, ListOfPreconditions)
```

For instance, we define the attack step which discovers the OTP generation function by looking at the XOR chaining:

```
attackStep(staticallyAnalyseOTPFunctionForCounter, [counterValue])
attackStepRequirement(staticallyAnalyseOTPFunctionForCounter,
          [OTPFunctionAreaLocated])
```

where `staticallyAnalyseOTPFunctionForCounter` is the name of the attack step that requires the presence of the `OTPFunctionAreaLocated` fact in the fact base and produces as output the `counterValue` fact.

[18]In backward programming, the word fact is used to denote all the information that is known and available for the reasoning

The function that determines all the attack paths is sketched here:

```
allAttackPaths(Goal, AttackPaths) :-
    initFactsBase(FactsBase),
    findall(AttackPath,
            getAttackPath(Goal, FactsBase, _, AttackPath),
            AttackPaths).
```

The `allAttackPaths` function initializes the fact base and generates all the attack paths that permit to achieve the goal passed as input by means of the `getAttackPath(Goal, FactsBase, _, AttackPath)` function. The entire recursive program is shown in Appendix F.

14.1.3 Modelling the application for attack path deduction purposes

Previous sections presented how to model attack steps (and the attack steps actually needed to model attacks against OTP) and how backward reasoning can be used to discover attack paths from individual attack steps. However, not all the application contain OTP generation functions, protected with a PIN, with a counter and a seed. That is, it is needed a precise characterization of the application to protect to limit the scope of backward reasoning and to identify the attack paths that can be actually used. That is, there are other things that need to be modelled for an automatic inference system to work correctly:

- The application has an application part, the `getCurrentOTP()` function, which generates OTPs, this application part uses two variables, the `seed` and the `counter`.
- The `seed` and the `counter` are assets, more precisely they are confidential private data (see D1.02).
- The `counter` value is hardcoded in the application at design time, that is, it is a fixed value.
- The application has an application part, the `getSeed()` function, which interacts with the server to receive the value of the seed variable.
- the `getSeed()` application part has an `executionCounter` variable, initialized to zero. The acceptable values of the `executionCounter` are only 0 and 1, that is, this variable serves to verify that the seed initialization is done only once.
- The application has an application part, the `checkPIN()` function, which performs user authentication by means of something the user knows. `checkPIN()` uses a variable named `pin`.
- The `pin` value is associated to the user and set in the application at provisioning time, that is, for our purposes it is a fixed value.

For future versions, we are considering the possibility to extend annotations semantics to add the application-specific knowledge that is needed to allow attack discovery. In any case, it will be possible to insert this information in the AKB by means of the ontology-driven XText-based tool we want to use for allowing backtracking (see Section 13.1).

From the application-specific knowledge stated before, it is possible to deduce more information:

- The server is able to notice when the `getSeed()` is executed more than once and detect it as an anomaly if the application has not been re-installed.
- Re-installing the application resets the `executionCounter` variable used by the `getSeed()` function.

- The execution of the `getSeed()` function resets the seed value.

Finally, other information can be obtained by a simple but focused code analysis:

- The `getCurrentOTP()` makes use of XOR function, a statically and dynamically recognizable pattern.
- The `getSeed()` function uses a cryptographic algorithm, the AES, in deciphering mode, a statically and dynamically recognizable pattern.

14.2 Deducing when a protection can be used

Once the data collection has been completed, the ADSS can begin to detect the protections that can be applied to a given set of assets. This information is primarily deduced by means of several ontology-centric modules and techniques.

In order to achieve a meaningful and useful set of deductions, the AKB must contains all the application-specific information acquired during the Data Collection phase (see Section 13.1) together with, of course, the a priori information:

- the list of meaningful attack tools, attacks, software protections and assets;
- what other tools a particular attack tool can use;
- what tools can be used to mount an attack;
- what attacks a specific software protection is able to mitigate;
- the attacks that can threaten a specific asset;
- the attacks that can invalidate a protection.

We give an informal description of the inferences that are computed by the ADSS, in particular if the anti-debugging and code flattening protections, which are researched in WP2 and whose preliminary specification is to some extent documented in D1.04, are suitable for a certain asset. Figure 22 sketches a simplified view of the deductions that the ADSS performs in the scenario that the applicability of an anti-debugging technique must be assessed. The dashed connections represent the inferred properties, while the solid ones are the relationships gathered previously during the data collection phase. The inference regarding the applicability of a protection technique works in several steps. From a reasoner point of view there is no need to define an order of execution, but for the sake of clarity, these stages are detailed and described in a particular order, which is easier to understand for humans.

The steps performed by the ADSS are the following.

1. Since the attack tool `attackTool1` makes use of the debugger `debugger1` it is classified as a `DebuggerBasedTool`.
2. The attack tool `attackTool2` makes use of the tool `attackTool1`, which uses a debugger, so it is also classified as a `DebuggerBasedTool`.
3. The attack `attack1` is classified as a `DebuggingAttack` attack because it uses `attackTool2`, which is a debugger-based tool.
4. The protection `protection1` is of type `antiDebuggingType` and since `attack1` is a debugging attack, it is inferred that `protection1` highly mitigates `attack1`.

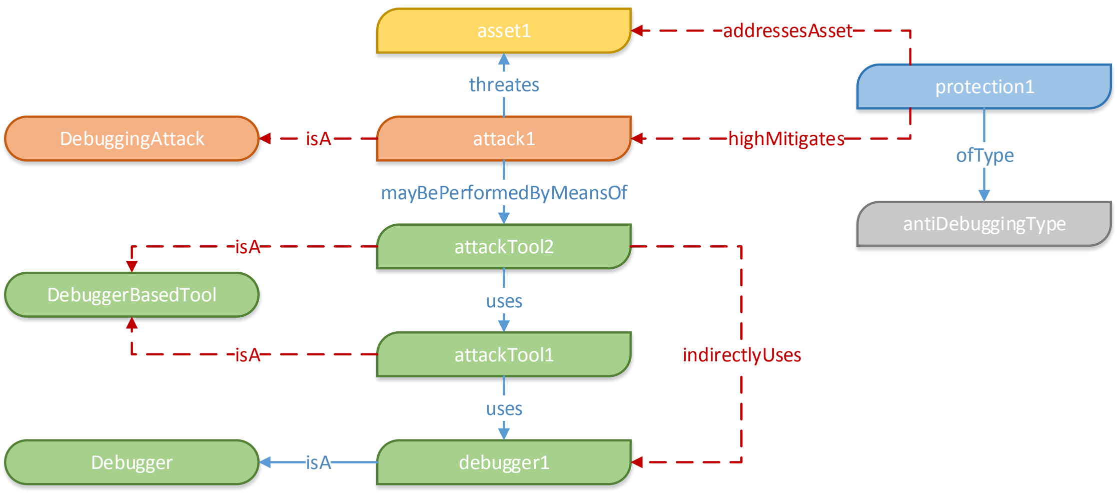

Figure 22: Anti-debugging inferences in a simplified scenario.

5. Since the `attack1` threatens the asset `asset1` and is mitigated by `protection1`, the ADSS deducts that `protection1` can be used to protect `asset1` against `attack1`.

Note that in addition to deducing that a specific software protection can mitigate an attack, the ADSS is also able to give a score (high, medium, low) to the protection level achieved against an attack[19]. For instance, the anti-debugging protection demonstrates the following levels of mitigation:

- high mitigation against tampering attacks;
- high mitigation against dynamic structure analysis;
- high mitigation against debugging-base attacks;
- high mitigation against flow analysis;
- low mitigation against structural matching of binaries performed using instruction pattern based tools.

Deductions to understand the need for using code flattening are very similar to the ones depicted in Figure 22.

Obviously, for the code flattening protections, the properties to represent the mitigated threats are different:

- high mitigation against flow analysis;
- medium mitigation against comparison of execution traces;
- low mitigation against structural matching of binaries performed using instruction pattern based tools.

By representing the facts related to the protection techniques and the axioms used to infer their applicability in the AKB, we noticed that a common modeling and reasoning pattern has emerged.

[19]These properties are very coarse grained classification of the mitigation levels that can be used to first select techniques and reduce the space of the combination generation and are not substitute of the metrics.

This seemingly general deduction strategy will be applied to model all the protection techniques that will be developed during the ASPIRE project.

Finally, we should make a remark on the methodology for defining the deductions in these examples. Most of the information presented in this section has been acquired via interviews with the *protection owners*, which are the experts in the ASPIRE Consortium in each of the analysed technique. They will be also in charge for developing the techniques and also provided protection descriptions in D1.04. Data acquired by means of these interviews have been subsequently translated into logical rules and facts into AKB. Finally, the deduction system has been validated by the same experts.

15 The ASPIRE Enrichment Framework

Section authors:
Daniele Canavese, Cataldo Basile

As anticipated in section 13.1, we will implement the AKB as a formal ontology represented in OWL-DL. There are several reasons to use an ontology to represent the AKB, these are mainly related to the intrinsic inferential properties of ontologies that (after our preliminary analysis, our past experience with other large projects, and analogous findings in related fields) will are perfect to address the problems ASPIRE has to tackle. Additionally, several tools exist to support the whole ontology life time, including editing, managing, and reasoning about the knowledge an ontology contains. An ontology can be thought of as a repository where a set of UML-like diagrams (with classes, instances and relationships) can be stored. Unlike UML, however, ontologies can also store other kind of facts (e.g., stating that a relationships is transitive) and can perform deductions over these collection of facts using a specialized software component called *reasoner*. The logic family behind the ontologies that we will use is known as *description logic* (DL), a very expressive system which is a fragment of the FOL, (i.e., the traditional logic commonly used in math). The main reason to use DL is that in several practical scenarios, DL reasoning algorithms have better performances than their FOL counterparts even if they have the expressiveness needed by most types of inferences that may be needed in practice.

Hence, some of the facts that can be written in the AKB consists of:

- equivalences and subsumptions[20] of classes and relationships;
- unions, intersections and negations of classes;
- use of the universal ($\forall$) and existential ($\exists$) quantifiers in the axioms;
- specification of a number of relationship characteristics (e.g., transitivity, symmetry, reflexivity, . . .);
- cardinality restrictions on relationships.

More information on description logics can be found in Section E.1.

Description logics traditionally make use of the so called *open world assumption*, under which ontologies assume that the specified set of facts *may* be incomplete. This approach has several repercussions on the kind of reasoning that can be made, by limiting their applicability in a number of scenarios. In these scenarios, the use of another logic system may be the best solution. This explains the need for the ADSS to make use of a number of external software components, the EMs, that allow the AKB to be optionally linked with some other external tool such as a theorem prover, a satisfiability (SAT) problem solvers or simply ad hoc tools implemented to perform a specific form of reasoning.

The ADSS will leverage various enrichment approaches in order to overcome the (semantical and computational) limitations of the ontologies and to further extend its capabilities as a powerful expert system. Therefore, it will use an Enrichement Framework.

The architecture of the enrichment framework has been already sketched in Figure 19.

The software components in charge of performing some kind of enrichment are called *enrichment modules*, or EMs for short. In a nutshell, each enrichment module will read some data from the AKB, will execute some custom code (with or without using other external reasoning tools) to deduce additional information and will write back the new data into the AKB.

[20]That are specializations of classes or relationships.

In order to simplify the usage of the enrichment framework to the users and developers, its public interface will be exposed through a special software module, the *Enrichment Coordinator*. Its tasks consist of executing the EMs, keeping track of their status and if needed, report their results to the user by making use of a GUIs and logs. Furthermore, the Enrichment Coordinator must be aware of the fact that an EM can rely on the results inferred by another EM, that is, EMs have dependencies. This means that the Enrichment Coordinator also has the task of establishing a correct ordering among the EMs based on their dependencies. For this purpose, each EM will be provided with an EM Manifest File, which will provide all the relevant data to allow the Enrichment Coordinator to correctly perform its job.

The precise definition of the EM Manifest File will be done during the next months, however, the EM Manifest File will contain at least information about how to call it (e.g., the name of plug-in to call), the list of the AKB classes it takes as input, the list of the AKB classes of the individuals [21] it writes in the ontology, and, if available, a list of explicit dependencies on other EMS that need to be executed before or after.

Enrichment modules can exhibit an astonishing number of differences due to their highly customizable nature. They can however be classified in several ways. From the ADSS user point of view they can be simply subdivided in:

Automatic EMs are non-interactive components that will be executed in a fully automatic fashion without asking anything to the user. Optionally, they can provide a validation report (through a GUI) to allow the user to check (and possibly discard part of) the deducted information before adding it to the AKB.

User interaction modules by contrast depend user intervention. For example, the reasoning performed by the enrichment framework can deduce that some required data is missing and not inferrable. Then the user is asked to provide these data via a GUI.

Another, more technical and sophisticated way of classifying the EMs is from a developer point of view, that is by distinguishing them by the technology that they use to perform the enrichment. Using this taxonomy we have basic modules, data outsourcing modules and ontology based modules.

15.1 Base modules

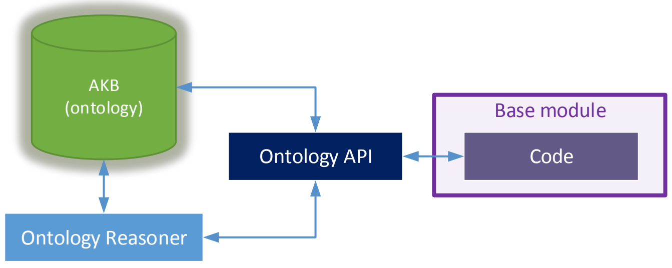

Figure 23: Base modules architecture.

The simplest kind of enrichment modules are known as *base modules*. Their architecture is shown in Figure 23. A base module communicates with the AKB using the AKB API to inspect and analyze the AKB content. Subsequently, the module will use some custom algorithm to infer

[21] We recall here that individual is the name used to indicate instances of ontology classes

new knowledge. Inferred data will be translated into a set of axioms and/or semantic rules that will be inserted in the AKB by means of the same AKB API. It is not required to add all the inferred knowledge at once, these modules may also add blocks of axioms as soon as they are available. Every time they add some axioms, these modules may require the activation of the ontology reasoner on the AKB to enrich it.

Their architectural simplicity however does not necessarily imply a limited possibility to perform complex reasonings, since the developer has at its disposal the power of a programming language for implementing custom inferences. Note that these enrichment modules do not rely on external tools or additional ontologies (which may in turn require the use of ontology reasoners). They perform their job by only making use of a traditional programming language without any 'external' support.

An example of base module we plan to develop is a graph-based attack-protection dependencies navigator. Starting from the application-specific knowledge and the a priori relations between attacks that can render unusable protections, and protections that can mitigate attacks, the graph-based inference (which is a form of reachability analysis) can be used to deduce the protections for which it is worth running the deduction of applicability (as discussed in Section 14.2). Graph-based inferences are easier to implement as an EM, as general-purpose languages have libraries that already code the most important and useful graph algorithms. Moreover, graph algorithms are usually much faster than ontology reasoning task.

15.2 Data outsourcing modules

DL techniques are powerful and flexible logical tools, but in some circumstances there may be a better logic systems (e.g., because of the Open World Assumption). A better logic system in ASPIRE means that the logic system permits the modelling of the same inferences in an easier or more accurate way, or that the logic system that can perform reasoning that is not possible in DL ontologies, or that the logic system uses a faster logic engine, that is, an engine that requires less time to infer the same data.

To enable this, the AKB can be interfaced with external tools using ad hoc EMs, which have been named *data outsourcing modules*. A data outsourcing module makes use of some external tools or libraries to gain new knowledge from other logic systems and engines. The architecture of data outsourcing modules is depicted in Figure 24. There exists a great variety of mature tools that are suitable for the needs of the ASPIRE project such as various forward and backward chaining logic engines, like Drools (http://www.drools.org/), Prolog (http://www.swi-prolog.org/), and graph-based tools, like graph-tool (http://en.wikipedia.org/wiki/Graph-tool).

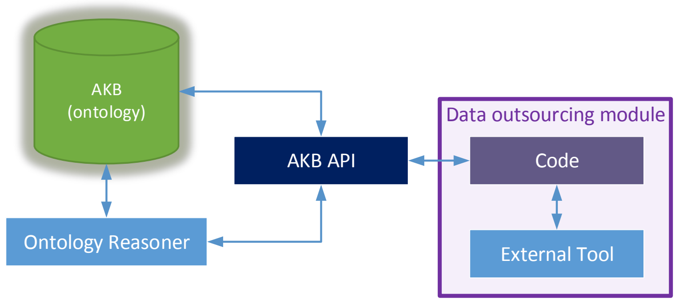

Figure 24: Data outsourcing module architecture.

An example of this EM type is the Prolog-based EM we will develop to deduce the attack paths from the attack steps, as explained in Section 14.1. Additionally, all the ACTC component will be implemented as data outsourcing modules when they have to insert protection profile information.

15.3 Ontology based modules

Ontologies are very extensible and expressive tools and, apart from being used for the AKB representation, they can be employed by ad-hoc *ontology based modules*. We can distinguish two kinds of ontology based EMs that have very different scopes and architectures.

The first type of ontology based modules are primarily used to reduce the reasoning time. The high expressivity of description logic can sometimes come at the cost of computational performance. Moreover, reasoning on large ontologies may have bad performance. *Ontology decentralization module*s can be effectively used to counteract these effects. The basic idea is to perform a traditional reasoning over a small or less expressive ontology (e.g., an OWL-EL or OWL-QL ontology) which represents a close and coherent domain of knowledge (e.g., a subset of the AKB), and then to copy the inferred axioms into the AKB. The architecture of these EMs is depicted in Figure 25. The deduction of the usability of a protection, shown in section 14.2, is a candidate type of reasoning to determine protections dependencies based on the a priori knowledge and on the application-specific data.

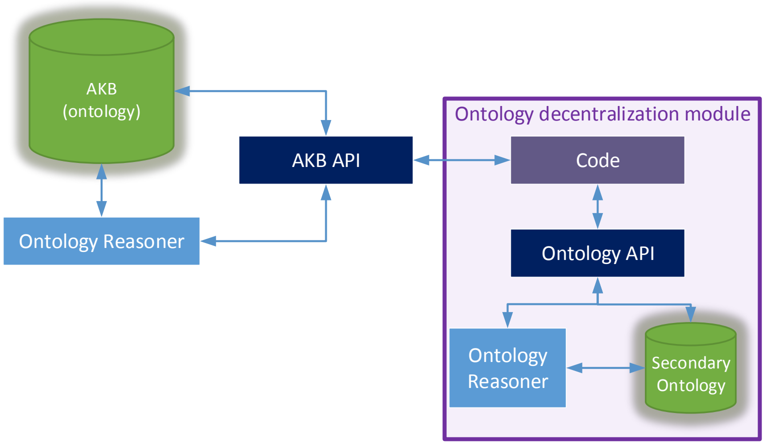

Figure 25: Ontology decentralization module architecture.

The second kind of ontology based modules are instead used to add expressivity, that is semantic, to an ontology. For this reason, they are called *semantic injection module*s. A generic semantic rule has the following structure:

$$P_1(\cdot) \vee P_2(\cdot) \vee P_3(\cdot) \vee \cdots \Rightarrow Q_1(\cdot) \vee Q_2(\cdot) \vee Q_3(\cdot) \vee \dots$$

The meaning of the rules is that if all the $P_i(\cdot)$ are true, then also the predicates $Q_i(\cdot)$ holds. The $P_i(\cdot)$ and $Q_i(\cdot)$ are essentially Boolean functions that operates on a set of ontology entities (i.e., individuals, classes and properties). Semantic rule languages come with a number of built-in predefined functions, but using a semantic injection module a new one can be 'injected' in the reasoner, thus increasing the expressivity of the rules. The architecture of these EMs is sketched in Figure 26.

In order to avoid closure and soundness problems, the semantic injection modules have some limitations. For instance, these EMs can only read data from the ontology; if a write is performed, an infinite loop can arise, since the reasoner will be triggered by the write and will execute the rule again, thus potentially generating another write.

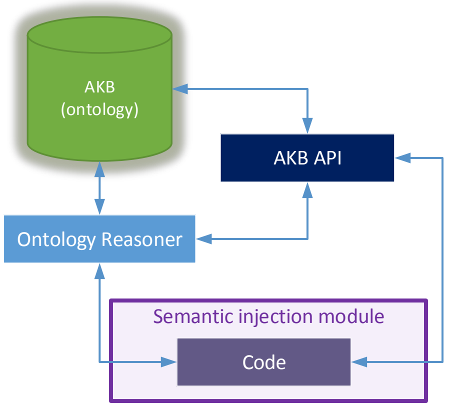

Figure 26: Semantic injection module architecture.

These EMs are very different from the base modules. The latter makes use of a reasoner and an ontology by exploring and altering it, the former are instead automatically called by the reasoner when needed.

Part IV

Appendices

A Data-Specific Annotations

Section authors:
Roberto Tiella, Cataldo Basile, Rachid Ouchary, Bjorn De Sutter, Bart Coppens, Alessandro Cabutto, Werner Dondl, Jerome D'Annoville, Brecht Wyseur

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>))
```

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

- Case `mask(constant(<INTEGER>))` : using the mask number specified;
- Case `mask(random(<INTEGER>,<INTEGER>))` : using a random mask number in the range specified.

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 both fields `x` and `y` belonging to the structure Point to be encoded using XOR with a random key chosen between 100 and 150:

```
struct Point {
  int x __attribute__((ASPIRE("protection(xor,mask(random(100,150)))")));
  int y __attribute__((ASPIRE("protection(xor,mask(random(100,150)))")));
};
```

A.2 Merge Scalar Variables

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

```
<PROTECTION_NAME> ::= merge_vars

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

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.
- size(<INTEGRAL_SIZE>): the number of bits allocated to the variable.

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),size(6))"))) = 0;
int y __attribute__((ASPIRE("protection(merge_vars,set(s1),size(2))"))) = 0;
```

A.3 Residue Number Coding

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

```
<PROTECTION_NAME> ::= rnc

<PROTECTION_PARAMETER> ::=   range(<INTEGER>,<INTEGER>)
                           | base(constant(<LIST_OF_INTEGERS>))
                           | base(random(<INTEGER>))
```

Semantics:

- range(<INTEGER>,<INTEGER>): specifies the dynamic range of values stored in the variable, i.e., the range of values that might be observed in the variable during the program's execution.
- base(constant(<LIST_OF_INTEGERS>)): specifies the sequence of pairwise prime integers used in the encoding.
- base(random(<INTEGER>)): specifies that the sequence of pairwise prime integers used in the encoding has to be chosen randomly. <INTEGER> specifies the sequence's length.

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)))")));
```

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(dato_to_proc,algorithm(mealy_lutable))")
     )) = "password";
```

A.5 Multi-threaded Cryptography

As described in Section 3.6 of deliverable D1.04, multithreading 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:

```
char * key __attribute__((ASPIRE("protection(multi_threaded_crypto,"
                                 "algorithm(AES),"
                                 "mode(CBC),"
                                 "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.
- `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 timebombs 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");
 }
```

B Code-Specific Annotations

Section authors:
Roberto Tiella, Cataldo Basile, Rachid Ouchary, Bjorn De Sutter, Bart Coppens, Alessandro Cabutto, Werner Dondl, Jerome D'Annoville, Brecht Wyseur

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"

static const char init_vector[]
     = {0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7};

char plaintext[64];

char* key;

char ciphertext[64];

init_plain_text(plaintext);
init_key(key);
```

```
wbgcClientEncryptExampleDynamic(plaintext,64,key,init_vector,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> ::= in ( softvm | application )
```

Semantics:

- Case `in (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 `in (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 `in (application)` will *never* be moved to the SoftVM.
- 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(in_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(in_application)");
12   for (int i = 0; i <bound; i++)
13     z+=x;
14   _Pragma("ASPIRE end");

15   return z;
16 }
```

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.

B.3 Multi-threaded Cryptography

As described in Section 3.6 of deliverable D1.04, we will use multithreading 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('VEhJU01TT05FQUVTS0VZOQ==')))")));
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 )
                         |  propagate_to_callees  ( enable | disable )
```

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.
- If `propagate_to_calls` is set to `enable`, the annotations will propagate to all callees in this code region. If it is set to `disable` (the default), the annotations do not propagate. If this results in conflicting annotations on certain code regions, the toolchain should abort.

```
 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 )
                         |   propagate_to_callees ( enable | disable )

<PROTECTION_NAME> ::= guard_attestator

<PROTECTION_PARAMETER> ::=
           | label ( <ID> )
           | regions (<LIST_OF_IDS>)
           | attest ( never )
           | propagate_to_callees ( enable | disable )


<PROTECTION_NAME> ::= guard_verifier

<PROTECTION_PARAMETER> ::=  attestator ( <ID> )
                         | propagate_to_callees ( enable | disable )
```

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.

If propagate_to_callees is set to enable, the annotations will propagate to all callee's in this code region. If it is set to disable (the default), the annotations do not propagate. If this results in conflicting annotations on certain code regions, the tool chain should abort.

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 numeric 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 obfusations 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 parametrized 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.

```
int dd1;
int dd2;
int year1;
int year2;

void g() {
  int y;
  f(y);
 }

void f(int x) {
  _Pragma("ASPIRE begin protection (barrier_slicing, barrier(year1),"
          "label(slicing1))")
  year1 = read();
  _Pragma("ASPIRE end")
  int ref = year1;
  int year2 = read();

  for (int i = ref; i < year1; i++) {
    if (i % 4 == 0)
       dd1 += 1;
    }
  dd1 = calculate_original();

  dd2 = 0;
  for (int i = ref; i < year2; i++) {
    if (i % 4 == 0)
        dd2 += 1;
    }
  dd2 = calculate_current();

  _Pragma("ASPIRE begin protection (barrier_slicing, criterion(dd1,dd2),"
          "label(slicing1))")
  if (dd2 - dd1 > 30)
    printf("Fail\n");
  else
    printf("Ok\n");
  _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 )
                        |    percent_to_mobile_blocks(<INTEGER>)
                        |    blocks_size(<INTEGER>))
```

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.
- `blocks_size`: mobile code blocks size in bytes. If this parameter is not specified, a random size can be chosen.
- `percent_to_mobile_blocks`: percentage amount of code to be transformed in mobile code.

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(1),block_size(512))");

  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>)
                         |   dynamic_ra(<LIST_OF_IDS>)
                         |   temporal_ra(<LIST_OF_IDS>)
                         |   implicit_ra(<LIST_OF_IDS>)
                         |   hashBased_ra(<LIST_OF_IDS>)
                         |   invariants_monitoring(<LIST_OF_IDS>)
                         |   SWATT_based_ra(<LIST_OF_IDS>)
```

Semantics: These annotations permit the specification of either a generic or a concrete remote attestation technique:

- `static_ra(<LIST_OF_IDS>)`: abstract type of RA. It indicates that the code must be protected with a static RA technique. In the long term, the decision of the concrete technique to use will be made by the ADSS, in the short term, the user can be prompted to select the technique to use;
- `dynamic_ra(<LIST_OF_IDS>)`: abstract type of RA. It indicates that the code must be protected with a dynamic RA technique. In the long term, the decision of the concrete technique to use will be made by the ADSS, in the short term, the user can be prompted to select the technique to use;
- `temporal_ra`, abstract type of RA. It indicates that the code must be protected with a temporal-based RA technique. In the long term, the decision of the concrete technique to use will be made by the ADSS, in the short term, the user can be prompted to select the technique to use;
- `implicit_ra(<LIST_OF_IDS>)`, concrete type of RA. It indicates that the code will be protected with the implicit remote attestation that will be developed in T3.2. The research on implicit attestation is ongoing, the parameters will be defined with the theoretical framework;
- `hashBased_ra(<LIST_OF_IDS>)`: concrete type of static RA. It receives as parameters the hash function to use and the function to compute (named attestation data generation function in D1.04).
- `SWATT_based_ra(<LIST_OF_IDS>)`: concrete type of temporal RA. It uses injects very optimized pieces of code that compute iterative hashing of data taken from a random walk in the memory pages and identifies violations as deviations in the request-response time [7]. It receives as input the hash function to use, the attestation data generation function;
- `invariants_monitoring`: concrete type of Dynamic RA. Asks the ACTC to deploy an invariant monitoring remote attestation. Invariant predicates are passed as parameters and need to be defined with following syntax:

```
<PROTECTION_PARAMETER> ::= invariant(<ID>=PREDICATE)

<PREDICATE>  ::= EXPRESSION, OP, CONSTANT;
<EXPRESSION> ::= TERM | EXPRESSION, "+" , TERM;
<TERM>       ::= FACTOR | TERM, "*" , FACTOR;
<FACTOR>     ::= CONSTANT | VARIABLE | "(" , EXPRESSION , ")";
<OP>         ::= "<", ">", "==", "!=", "<=", "<="
```

This initial invariant definition permits to define invariants as predicates that compare polynomials to constant values. This definition will be updated during the next months to support, if needed, other more complex functions. The variables used by the invariants need to be tagged using data annotations to avoid ambiguities. For instance, the following annotation is used to tell that variable x is used by invariant monitoring in the invariant named Invariant1 where it is named var:

```
      int x __attribute__((ASPIRE("protection(invariant_monitoring,
                                              Invariant1,var)"))) ;
```

Example: This example shows how the developers can ask the framework to protect a code region using invariants monitoring RA. Only one invariant is defined, which states that the sum of the values of variables x and y is less than 100. Note that, variables used in the invariant have been tagged and associated to the invariant where they are used.

```
int f(int max) {
  int i,sum = 0;
  int y __attribute__((ASPIRE("protection(remote_attestation,InvariantXY)")));
  _Pragma("ASPIRE begin protection(remote_attestation,
                          invariants_monitoring,invariant(InvariantXY=x+y<100))");

  for(i=0; i<max; i++){
     y=2*max;
     sum+=y;
  }
  _Pragma("ASPIRE end");

  return sum;
}

int main()
{
   int x __attribute__((ASPIRE("protection(remote_attestation,InvariantXY,x)")));
   x = 33;
   printf("Sum=%d",f(x));
   return 0;
}
```

B.11 CFG Tagging

The tamper detection technique of CFG tagging 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 CFG tagging 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> ::= cfg_tagging

     <PROTECTION_PARAMETER> ::= gate(<ID>)
                              | <VERIFIER>
```

```
<VERIFIER> ::= check (<BOOL_EXPRESSION>)
             | location ( <LOCATION> )

<BOOL_EXPRESSION> ::= <SQ_STRING>

<LOCATION>  ::= local | remote
```

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 specifies which gates must have been activated or not. 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. The value of the counter can be retrieved in the expression with the following operator: ID.counter; 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.

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(cfg_tagging,gate(main_if))");
  //...
  _Pragma("ASPIRE end");
}
else
{
  _Pragma("ASPIRE begin protection(cfg_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(cfg_tagging,"
         "check('(main_if+main_else)*function1)'),location(local))");

//...
```

```
  _Pragma("ASPIRE end");

void function1()
{
  // ...
  _Pragma("ASPIRE begin protection(cfg_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*. 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 developper to specify on the timebombs 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
```

C Annotation Fact Example

Section authors:
Mariano Ceccato

Considering the example of annotated code in Figure 27, we have two annotations in line 9 and line 23, respectively closing at line 11 and 23. The corresponding JSON file is shown in Figure 28. Each fact file consists of an array of objects, one object per annotation.

```
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")
      }
```

Figure 27: Example of source code with annotations.

```
[{
  "file name":"example.i",
  "line number":[9,11],
  "annotation type":"code",
  "function name":"f",
  "annotation content":
         "protection (barrier_slicing, barrier(year1), label(slicing1))"
 }, {
  "file name":"example.i",
  "line number":[23,28],
  "annotation type":"code",
  "function name":"f",
  "annotation content":"protection (barrier_slicing, criterion(dd1,dd2),
  label(slicing1))"
 } ]
```

Figure 28: Example of annotation fact in JSON.

D JSON Format for Diablo - X-translator Interface

Section authors:
Bjorn De Sutter, Sander Bogaert, Jonas Maebe, Andreas Weber

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")
   "encoding" : string (hexadecimal representation of encoded instruction)
   "regswritten" : array of strings (e.g. ["R0", "R2"])
   "regsread" : array of strings
   "addrsymbol" : integer (index in exported symbols array)
}

basic_block {
   "instructions" : array of instruction objects
   "regsliveout" : array of strings
}

edge {
   "sourcebbl" : integer (id = position of bbl in "bbls" array)
   "destbbl" : integer (id = position of bbl in "bbls" array)
   "destsymbol" : integer (id = position of symbol in symbols array)
   "type" : string ("jump", "fallthrough", "call", "return", "switch")
}

chunk {
   "bbls": array of basic_block objects
   "edges": array of edge objects
   "regslivein": array of strings
}

symbol {
   "name" : string
   "value" : number
   "fileoffset" : number
}

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"]
    }
  ]
}
```

E Technical background on description logics and linear programming

Section authors:
Daniele Canavese, Rachid Ouchary, Cataldo Basile

This section presents a brief introduction of description logics and linear programming is presented.

E.1 Description logics

We present here more details on DL ontologies that will be used for the AKB implementation.

E.1.1 TBoxes and ABoxes

Description logics (DL) are a decidable fragment of first order logic (FOL) used to represent information and are widely adopted for terminological knowledge modeling [1]. In these cases, they provide a formal and non-ambiguous semantics to the terms of the discourse.

The principal descriptive means in DL are *concepts*, which can be inductively defined on the basis of a set of concept constructors. The variety of the allowed concept constructors, and the choice of which constructors to allow in a specific model, determine the specific description logic. The simplest entities are the atomic concepts and *roles*, denoting respectively domain sub-sets and relationships between domain elements. Starting from these elements, the complex concepts can be defined applying the constructors of the specific adopted logic. For example, Table 4 presents the concept constructors for the description logic used in the OWL ontology language.

OWL keyword	DL syntax	FOL syntax
`intersectionOf`	$C \sqcap D$	$C^I \cap D^I$
`someValuesFrom`	$\exists R.C$	$\{ a \in \Delta^I : \exists < a, b > \in R^I \land b \in C^I \}$
`allValuesFrom`	$\forall R.C$	$\{ a \in \Delta^I : < a, b > \in R^I \Rightarrow b \in C^I \}$
`unionOf`	$C \sqcup D$	$C^I \cup D^I$
`complementOf`	$\neg C$	$\Delta^I \setminus C^I$
`maxCardinality`	$\leq nR$	$\{ a \in \Delta^I : \sharp\{ b : < a, b > \in R^I \} \leq n \}$
`minCardinality`	$\geq nR$	$\{ a \in \Delta^I : \sharp\{ b : < a, b > \in R^I \} \geq n \}$
`cardinality`	$= nR$	$\{ a \in \Delta^I : \sharp\{ b : < a, b > \in R^I \} = n \}$
`oneOf`	$\{ a \}$	$\{ a \}$

Table 4: OWL concept constructors.

A *knowledge representation* system (KR) provides the means to define, manage, manipulate and reason on a *knowledge base* (KB) containing the available information. While using a DL approach, a knowledge base is split in two basic components: the TBox and the ABox.

The *TBox* (Terminology Box) introduces the terminology, which is the set of (both atomic and complex) concepts and roles described by using a number of logic formulas. Traditionally, the atomic concepts are represented with the letters A and B, the letters C and D are used to denote the complex ones, while R and S designates the roles. The terminology box contains a collection of axioms which can be distinguished in:

- *inclusion axioms* (or subsumptions), in the form $C \sqsubseteq D$ and $R \sqsubseteq S$;
- *equivalence axioms* (or definitions), in the form $C \equiv D$ and $R \equiv S$.

An simple but important property is that

$$C \equiv D \Leftrightarrow C \sqsubseteq D \wedge D \sqsubseteq C$$
$$R \equiv S \Leftrightarrow R \sqsubseteq S \wedge S \sqsubseteq R$$

The *ABox* (Assertional Box) contains factual extensional knowledge about the state of affairs in a specific model. Typically, the lower-case letters such as a, b and c are used to indicate the instances. The assertional box contains the following type of axioms:

- *concept assertions* (or class assertions), in the form $C(a)$;
- *role assertions* (or property assertions), in the form $R(a, b)$.

A knowledge base is interpreted through the classic extensional semantics. In this context, the knowledge base interpretation I is a couple $I =< \Delta^I, \cdot^I >$ of an interpretation domain Δ^I containing all the entities in the model and an interpretation function $\cdot^I$, which associates:

- to each individual a an element $a^I \in \Delta^I$;
- to each concept C a sub-set $C^I \subseteq \Delta^I$;
- to each role R a pair $R^I \subseteq \Delta^I \times \Delta^I$.

Given a particular knowledge base $KB =< T, A >$, where $T = \{ C_1 \sqsubseteq D_1, \dots \}$ is a TBox and $A = \{ C_1(a_1), \dots, R_1(b_1, c_1), \dots \}$ is an ABox, an interpretation I is said to be a *model*:

- of T if and only if $C_i^I \subseteq D_i^I, \forall i$;
- of KB if and only if I is a model of T, $a_i^I \in C_i^I, \forall i$ and $< b_i^I, c_i^I > \in R_i^I, \forall i$.

E.1.2 RBoxes

In some DL systems, a superior expressivity can be achieved by allowing the use of semantic rules. The simplest variant of *semantic rules* (or trigger rules) are expressions in the form

$$C \Rightarrow D$$

The meaning of this rule is that 'if an individual is proved to be an instanced of C, then it can be inferred that it is also an instance of D'. At first glance, this rule can be also interpreted as a synonym for $C \sqsubseteq D$. This interpretation is however wrong. Infact, usually, semantic rules are not equivalent to their contrapositives. For instance, $C \sqsubseteq D \Leftrightarrow \neg D \sqsubseteq \neg C$, but $C \Rightarrow D \not\Leftrightarrow \neg D \Rightarrow \neg C$. In addition, semantic rules can only be applied to the known objects in the KB, while traditional declarative axioms can also be applied to unknown instances. For example, $C \sqsubseteq D$ and $\neg C \sqsubseteq D$ implies that every object belongs to D, but $C \Rightarrow D$ and $\neg C \Rightarrow D$ can only be applied to an individual a for which either $C(a)$ or $\neg C(a)$ can be proven.

We can then extend the notion of a knowledge base by including these rules, thus having $KB =< T, A, R >$, where R is the *RBox* (rule box), that is the set of all the semantic rules in the knowledge representation system.

It can be proved that, under certain conditions, the RBox assertions can be transformed in ABox assertions, so that we can transform a knowledge base $KB =< T, A, R >$ into another equivalent one $\tilde{KB} =< T, \tilde{A} >$.

E.1.3 Reasoning

The reasoning services provided by a knowledge management system can be classified in ABox and TBox services, depending on the kind of knowledge involved in the task. The TBox services only make use of the schema level axioms to derive all the corresponding entailments and include:

- *subsumption checks* ($T \models C \sqsubseteq D$), where it is checked if a concept C is subsumed by another concept D;
- *TBox consistency checks* ($T \models \bot$), where it is verified if the TBox is satisfiable (or consistent), i.e., it is checked if there exists at least one model of T;
- *concept satisfiability checks* ($T \models C \not\equiv \bot$), where the satisfiability of a concept C is tested, i.e., it is verified if there exists at least one model I of T such that $C^I \neq \emptyset$.

The ABox services, on the contrary, elaborate in the same time both the factual extensional knowledge and the schema level intensional one. They are used to query the system for some information specific to a particular model:

- *consistency checks* ($KB \models \bot$), where it is verified that the whole knowledge base is satisfiable (or consistent), i.e., it is checked if there exists at least one model of the knowledge base KB;
- instance checks ($KB \models C(a)$), where it is tested if a given individual a is an instance of a specific concept C, limiting out the exploration to the interpretations that are models of the knowledge base;
- *instance retrievals* ($KB \models C(?)$), which is a non-trivial composition of other instance retrieval tasks. The system is queried for all the individuals that can be proved to belong to a given concept C.

Traditionally, the reasoning algorithms that works on a DL system can be split in two major categories:

- *structural subsumption algorithms* are very efficient, but they are complete only for rather simple languages with a significant lack of expressivity. They essentially rewrites every axiom in a subsumption (sub-classing) and tries to perform the inferences by comparing the axioms' syntactic structure;
- *tableau algorithms* instead exchange their computational efficiency for an increased support of DL families. These algorithms tries to infer new data by transforming every deductions into a concept satisfiability check, thus checking if a concept can or cannot contain at least one instance.

The modern reasoners, such as Pellet[22] and HermiT[23], usually belong to the latter algorithm category.

E.2 Linear programming

In this section we assume that the reader has familiarity with Mathematical Optimization therefore we will recall briefly the fundamental notions.

[22] http://clarkparsia.com/pellet/
[23] http://hermit-reasoner.com/

Linear programming is a mathematical modeling technique used in wide variety of applications, including scheduling, resource allocation, business planning and many other optimization applications). The basic idea of linear programming is to find the maximum or minimum under linear constraints. A Linear Program is composed of

- variables
- constraints
- objective function

where the variables take on numerical values, the constraints are used to limit the the values to a feasible region, and must be linear in the variables on the other hand the objective function defines which particular assignment of feasible values to the variables is optimal, it's one that maximize or minimize the objective function, the objective function must also be linear on the variables.

In general a linear program in any problem of the form:

$$\begin{aligned} &\max c^T x \text{ or } \min c^T x \\ &Ax \leq b \\ &x \geq 0 \end{aligned}$$

Where x represents the vector of variables to be determined, c and b are vectors of known coefficients[24] and A is a known matrix of coefficients. The values of A are often called *weights*. The expression to be maximized or minimized $c^T x$ is called the *objective function*.

Integer linear programming (ILP) is a linear program with the added restriction that some, but not necessarily all, of the variables must be integer valued. There are three different types of ILPs:

different variants of Linear Programming exist:

- integer Linear programming (ILP): in this class of problems all variables are required to be integer
- mixed integer programming programming (MILP): in this case some variables are restricted to be integer and some are not
- (pure) 0-1 linear programming: where all the variables are integers and are restricted to either zero or one (e.g., they are boolean variables). It is used in problems where a yes-no decision is to be taken regarding multiple choices.

In Aspire beside to the single objective optimization techniques, multiple objective optimization will be used. Multi-objective optimization considers different conflicting objectives simultaneously. In this case, there is no single optimal solution, but a set of optimal solutions with different trade-offs, called Pareto optimal solutions, or non-dominated solutions, and only one of these solutions is to be chosen. The process of solving a multi-objective optimization problems involves two main steps, identify the pareto optimal solutions, and choose a single preferred solution.

[24]The symbol c^T represents the vector transpose of c.

F Prolog attack path deduction example: support material

Section authors:
Cataldo Basile

This section presents the Prolog code that computes all the attack paths:

```
/*
Module is required in order to limit cluasoles usage in other SWI—Prolog
files using this module (use_module clause) and in Java applications.
*/
:- module(attackPathsFromAttackSteps,
          [allAttackPaths/2,
          getAttackPath/3
          ]).

:- use_module(otpInput).

%Query required to get an asset starting from the AttackPath's FinalFactsBase
getAsset(FinalFactsBase, Asset) :-
    assetHasProperty(ListAssetProperty, Asset),
    forall(member(RequiredFact, ListAssetProperty),
           member([RequiredFact], FinalFactsBase)).

%Query in order to get a list of attack paths to reach Goal
allAttackPaths(Goal, AttackPaths) :-
    initFactsBase(FactsBase),
    findall(AttackPath,
            getAttackPath(Goal, FactsBase, _, AttackPath),
            AttackPaths).

getAttackPath(Goal, OutputFactsBase, AttackPath) :-
    initFactsBase(FactsBase),
    getAttackPath(Goal, FactsBase, OutputFactsBase, AttackPath).

%FactsBase is input, NewFactsBase is output.
getAttackPath(Goal, FactsBase, NewFactsBase, [AttackStep|Tail]) :-
    attackStep(AttackStepID, ListProductedFacts),
    member(Goal, ListProductedFacts),
    isEnabledAttackStep(AttackStepID, FactsBase),
    updateFactsBase(FactsBase, AttackStepID, NewFactsBase),
    AttackStep = AttackStepID,
    Tail = []. %Tail must to be empty for java processing

getAttackPath(Goal, FactsBase, NewFactsBase, [Head|Tail]) :-
    attackStep(AttackStepID, ListProductedFacts),
    member(Goal, ListProductedFacts),
    \+isEnabledAttackStep(AttackStepID, FactsBase),
    attackStepRequirement(AttackStepID, RequiredFacts),
    length(RequiredFacts, Len),
    (Len > 1
    ->
        getAttackPaths(RequiredFacts, FactsBase, UpdatedFactsBase, Tail),
        updateFactsBase(UpdatedFactsBase, AttackStepID, NewFactsBase),
        Head = AttackStepID
    ;
        member(Fact, RequiredFacts),
        getAttackPath(Fact, FactsBase, UpdatedFactsBase, Tail),
        (\+member(AttackStepID, Tail) -> Head = AttackStepID ; true),
        updateFactsBase(UpdatedFactsBase, AttackStepID, NewFactsBase)).

getAttackPaths(RequiredFacts, FactsBase, NewFactsBase, Tail) :-
    getAttackPathForAll(RequiredFacts, FactsBase, NewFactsBase, Output),
    once(formatPaths(Output, Out)),
    reverse(Out, OutRev),
    once(appendChain(OutRev, Tail)).

getAttackPathForAll([], FactsBase, FactsBase, []). %Stop recursion.

getAttackPathForAll([Fact|Others], FactsBase, NewFactsBase, [Path|Paths]) :-
    getAttackPath(Fact, FactsBase, UpdatedFactsBase, Path),
    getAttackPathForAll(Others, UpdatedFactsBase, NewFactsBase, Paths).

%Functions.
%Check if attack step is enabled.
isEnabledAttackStep(AttackStepIdentifier, FactsBase) :-
    attackStepRequirement(AttackStepIdentifier, RequirementList),
    forall(member(Fact, RequirementList), containFact(Fact, FactsBase)).

%Check is Fact is in the FactsBase
containFact(Fact, FactsBase) :-
    member(ListFacts, FactsBase),
    member(Fact, ListFacts), !.

getAllInitAttackStep(InitAttackStep) :-
    findall(AttackStep,
            attackStepRequirement(AttackStep, []),
            InitAttackStep).

%FactsBase is a set of lists.
initFactsBase(FactsBase) :-
    getAllInitAttackStep(InitAttackStep),
    initialFactsBase(InitAttackStep, FactsBase).

initialFactsBase([], []). %Stop recursion.

initialFactsBase([Head|Tail], [HeadFB|TailFB]) :-
    findall(Fact,
            (attackStep(Head, ListProducted),
             member(Fact, ListProducted)),
             FactsToAdd),
    HeadFB = FactsToAdd,
    initialFactsBase(Tail, TailFB).

%FactsBase updated in NewFactsBase
updateFactsBase(FactsBase, AttackStepID, NewFactsBase) :-
    isEnabledAttackStep(AttackStepID, FactsBase) ->
        attackStep(AttackStepID, ProductedFacts),
        append(FactsBase, [ProductedFacts], NewFactsBase)
    ; fail.

formatPaths([], []). %Stop recursion.

formatPaths([Head|Tail], [RHead|RTail]) :-
    (isList(Head) ->
        append([], Head, RHead);
        append([], [Head], RHead)),
    formatPaths(Tail, RTail).

formatPaths(Element, List) :-
    append([], [Element], List).

appendChain([], []). %Stop recursion.

appendChain([[]|TailL], List) :-
    appendChain(TailL, List).

appendChain([[Elem|Others]|ListsT], [Elem|Tail]) :-
```

```
    appendChain([Others|ListsT], Tail).

appendChain([Element], [Element|[]]).

isList([_|_]).

isList([]).
```

G Ontology-based technique applicability deduction: support material

Section authors:
Daniele Canavese

This section lists the ontology axioms that are used by the AKB to compute the applicability of the anti-debugging and code flattening software protections, as explained in Section 14.2.

G.1 Anti-debugging axioms

Class axioms:

$$
\begin{aligned}
&Attack \sqsubseteq AttackStep\\
&Tampering \sqsubseteq Attack\\
&DynamicStructureAnalysis \sqsubseteq Attack\\
&DebuggingAttack \sqsubseteq Attack\\
&DebuggingAttack \equiv Attack \sqcap\\
&\qquad \exists mayBePerformedByMeansOf.DebuggerBasedTool\\
&StructuralMatchingOfBinaries \sqsubseteq Attack\\
&InstructionPatternBasedStructuralMatchingOfBinaries \equiv\\
&\qquad StructuralMatchingOfBinaries \sqcap\\
&\qquad \exists mayBePerformedByMeansOf.InstructionPatternBasedTool\\
&DynamicStructureAnalysis \sqsubseteq Attack\\
&FlowAnalysisAttack \equiv Attack \sqcap\\
&\qquad \exists mayBePerformedByMeansOf.FlowAnalysisBasedTool\\
&DebuggerBasedTool \equiv AttackTool \sqcap \exists indirectlyUses.Debugger\\
&Debugger \sqsubseteq DebuggerBasedTool\\
&FlowAnalysisBasedTool \equiv AttackTool \sqcap \exists indirectlyUses.FlowAnalysisTool\\
&FlowAnalysisTool \sqsubseteq FlowAnalysisBasedTool\\
&InstructionPatternBasedTool \equiv AttackTool \sqcap\\
&\qquad \exists indirectlyUses.InstructionPatternTool\\
&InstructionPatternTool \sqsubseteq InstructionPatternBasedTool
\end{aligned}
$$

Property axioms:

$$
\begin{aligned}
&uses \sqsubseteq indirectlyUses\\
&transitive(indirectlyUses) = true\\
&highMitigates \sqsubseteq mediumMitigates\\
&mediumMitigates \sqsubseteq lowMitigates\\
&lowMitigates \sqsubseteq mitigates
\end{aligned}
$$

Semantic rules:

$$
\begin{aligned}
&SWProtection(?protection) \wedge ofType(?protection, ?type) \wedge AntiDebug(?type) \wedge \\
&\quad Tampering(?attack) \rightarrow highMitigates(?protection, ?attack) \\
&SWProtection(?protection) \wedge ofType(?protection, ?type) \wedge AntiDebug(?type) \wedge \\
&\quad DynamicStructureAnalysis(?attack) \rightarrow highMitigates(?protection, ?attack) \\
&SWProtection(?protection) \wedge ofType(?protection, ?type) \wedge AntiDebug(?type) \wedge \\
&\quad DebuggingAttack(?attack) \rightarrow highMitigates(?protection, ?attack) \\
&SWProtection(?protection) \wedge ofType(?protection, ?type) \wedge AntiDebug(?type) \wedge \\
&\quad InstructionPatternBasedStructuralMatchingOfBinaries(?attack) \rightarrow \\
&\quad lowMitigates(?protection, ?attack) \\
&SWProtection(?protection) \wedge ofType(?protection, ?type) \wedge AntiDebug(?type) \wedge \\
&\quad DynamicStructureAnalysis(?attack) \rightarrow highMitigates(?protection, ?attack) \\
&SWProtection(?protection) \wedge ofType(?protection, ?type) \wedge AntiDebug(?type) \wedge \\
&\quad FlowAnalysisAttack(?attack) \rightarrow highMitigates(?protection \wedge ?attack) \\
&SWProtection(?protection) \wedge mitigates(?protection, ?step) \wedge AttackStep(?step) \\
&\quad threatens(?step, ?asset) \wedge Asset(?asset) \rightarrow addressesAsset(?protection, ?asset)
\end{aligned}
$$

G.2 Code flattening axioms

Class axioms:

$$
\begin{aligned}
&Attack \sqsubseteq AttackStep \\
&StructuralMatchingOfBinaries \sqsubseteq Attack \\
&InstructionPatternBasedStructuralMatchingOfBinaries \equiv \\
&\quad StructuralMatchingOfBinaries \sqcap \\
&\quad \exists mayBePerformedByMeansOf.InstructionPatternBasedTool \\
&DynamicStructureAnalysis \sqsubseteq Attack \\
&FlowAnalysisAttack \equiv Attack \sqcap \\
&\quad \exists mayBePerformedByMeansOf.FlowAnalysisBasedTool \\
&ComparisonOfExecutionTraces \sqsubseteq Attack \\
&FlowAnalysisBasedTool \equiv AttackTool \sqcap \exists indirectlyUses.FlowAnalysisTool \\
&FlowAnalysisTool \sqsubseteq FlowAnalysisBasedTool \\
&InstructionPatternBasedTool \equiv AttackTool \sqcap \exists indirectlyUses.InstructionPatternTool \\
&InstructionPatternTool \sqsubseteq InstructionPatternBasedTool
\end{aligned}
$$

Property axioms:

$$
\begin{aligned}
&uses \sqsubseteq indirectlyUses \\
&transitive(indirectlyUses) = true \\
&highMitigates \sqsubseteq mediumMitigates \\
&mediumMitigates \sqsubseteq lowMitigates \\
&lowMitigates \sqsubseteq mitigates
\end{aligned}
$$

Semantic rules:

$$
\begin{aligned}
&SWProtection(?protection) \wedge ofType(?protection, ?type) \wedge CodeFlattening(?type) \wedge \\
&\quad InstructionPatternBasedStructuralMatchingOfBinaries(?attack) \rightarrow \\
&\quad lowMitigates(?protection, ?attack) \\
&SWProtection(?protection) \wedge ofType(?protection, ?type) \wedge CodeFlattening(?type) \wedge \\
&\quad ComparisonOfExecutionTraces(?attack) \rightarrow mediumMitigates(?protection, ?attack) \\
&SWProtection(?protection) \wedge ofType(?protection, ?type) \wedge CodeFlattening(?type) \wedge \\
&\quad FlowAnalysisAttack(?attack) \rightarrow highMitigates(?protection, ?attack) \\
&SWProtection(?protection) \wedge mitigates(?protection, ?step) \wedge AttackStep(?step) \\
&\quad threatens(?step, ?asset) \wedge Asset(?asset) \rightarrow addressesAsset(?protection, ?asset)
\end{aligned}
$$

H ADSS specifications

Section authors:
Cataldo Basile, Daniele Canavese, Rachid Ouchary

This section presents the initial specifications of the ADSS architecture and API that will be used for the ADSS implementation. Clearly, both the architecture and the APIs may change in the next months, when the actual implementation will start. The ADSS specification is also important because it is the way to allow third parties to contribute to the ASPIRE project. It also an asset for exploitation purposes.

The general architecture of the ADSS is presented in Figure 29, while the next subsections present the details of each component. It is possible to see how an ADSS Workflow Coordinator, which implements the workflow presented in section 13.1, and all the components already listed in section 13.2, that is, the Data Collector, the Protection Combination Generator, the ADSS Pruning Module, the ADSS Optimization Module, the ToolFlow Bridge Module, and the AKB.

The ADSS Workflow Coordinator also exposes the main ADSS API (public API). Additionally, it also exposes a method available to its internal components (private API).

ADSS API

public API:

- `setContext(params)`: sets the context passing the required parameters (including the application source code and the preferences) to select the golden combination. If some of the required parameters are not passed in the context, they will be asked through a GUI (e.g., the link to the application source code);
- run(): executes all the phases to get the golden combination and produce the report;
- `getToolFlowInstructions()`: outputs the instructions to be passed to the tool flow to deploy the golden combination;
- `getAnnotatedSourceCode()`: outputs the application source code that includes the annotations generated by the ADSS to enforce the golden combination;
- `getReport()`: outputs the security report which include the golden combination;

private API:

- `notifyTermination(moduleID)`: used to coordinate the internal component workflow, this method serves to notify the termination of a module with ID "moduleID".

H.1 ASPIRE Knowledge Base (AKB)

Description of the component:

This component is responsible to provide access to the ASPIRE Knowledge Base. It wraps the actual repository by means of a limited CRUD API which only provides the operations that are considered safe. Read is considered safe as well as adding new individuals, adding new object or data properties. On the other hand, all the edit and delete methods are unsafe. Only individuals can be deleted. During the ASPIRE project the repository will be an OWL-DL ontology, nevertheless, other developers are allowed to change the internal representation (e.g., a plain DB) provided they match the API.

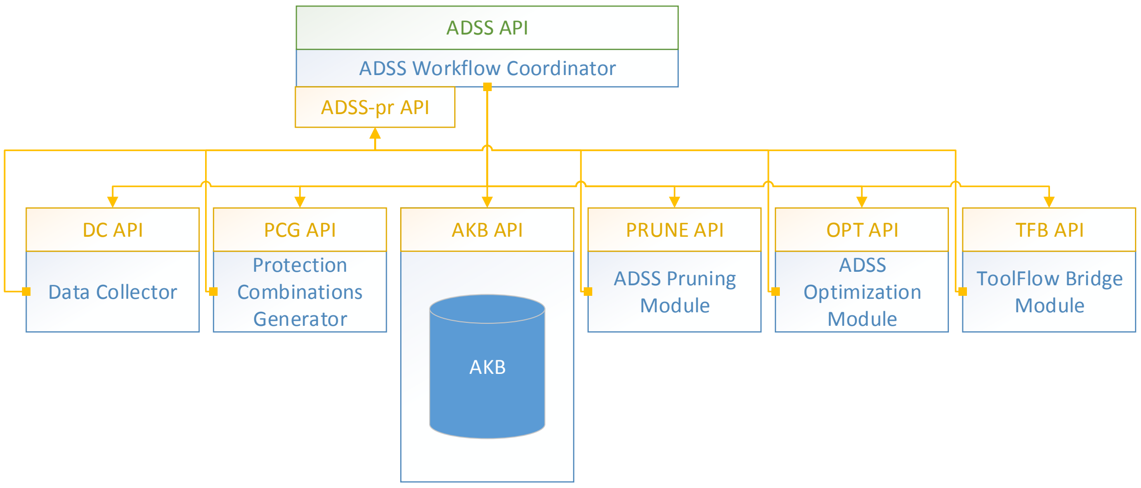

Figure 29: General ADSS architecture

AKB API:

- `createIndividual(class, name)`
- `deleteIndividual(name)`
- `addObjectProperty(individual1, individual2, objectProperty)`
- `addDataProperty(individual, dataProperty, value)`

H.2 Data Collector (DC)

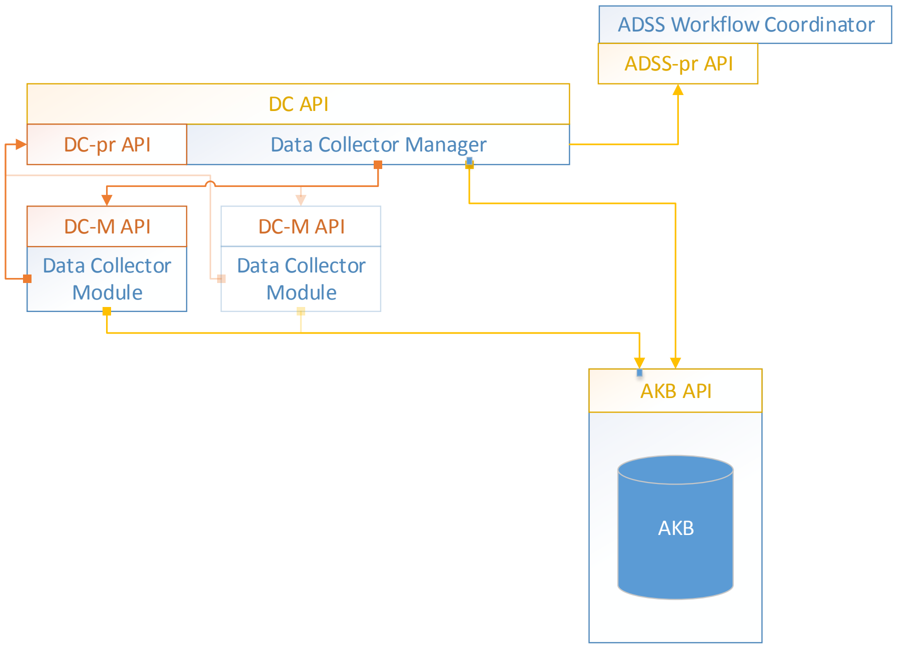

Figure 30: DC Architecture

Description of the component:

This component is responsible to perform the Data Collection phase of the ADSS workflow (see Figure 30). It is composed of a Manager and several Data Collection Modules, each one performing a specific task. For instance, the compiler tools which analyses the annotations in the application source code will be implemented as a Data Collection Module, as well as each GUI to collect inputs from users. Each Data Collection Module will be provided with a Data Collection Module Manifest File which is used by the Manager to determine how and when to call it.

This component also calls the ADSS `notifyTermination()` method when it completes its task.

DC API

- public API:
  - `setContext(contextParams)`: sets the context passing the required parameters to perform the data collection task
  - `run()`: decides the right data collector modules to call, based on the context information, then executes them;
  - `commit()`: commits the data collection results to the ontology (can be useful after a validation report is shown to the user)
- private API:
  - notifyTermination(DC_moduleID): used to coordinate the internal component workflow, this method serves to notify the termination of a module with ID "moduleID"

Data Collector Module (DC-M) API:

- performDataCollection(Params): called by the DC to gather some data that will be inserted in the AKB

H.3 Protection Combination Generator (PCG)

Description of the component:

This component performs the Combination Generation phase of the ADSS workflow (see Figure 31), that is, it generates the suitable combinations for one or more asset in the application to protect. It is composed by the AKB Enrichment Framework component, which is in charge to add new knowledge in the AKB by means of several inference techniques, and by the Combination Extractor component, which is in charge to actually extract and store in the AKB the suitable combination from the enriched AKB. The enrichment framework is composed of several Enrichment Modules (EMs), each one performing a specific inference task (in a given portion of the AKB) and identified by an ID, which are coordinated by an Enrichment Coordinator. Enrichment modules are provided with an EM Manifest File which is used by the Enrichment Manager to determine how and when to call them. The types of Enrichment Modules are listed in section 15. Enrichment modules do not automatically add into the AKB all the information they infer. This design requirement is introduced to permit the user to validate inferred data before inserting them into the AKB (e.g., through an EM validation GUI). For what concerns the extraction, the Combination Query Engine queries the AKB to generate the suitable combinations. Additionally, the Combination Filter Engine is in charge to generate directives to limit the generation of suitable combinations

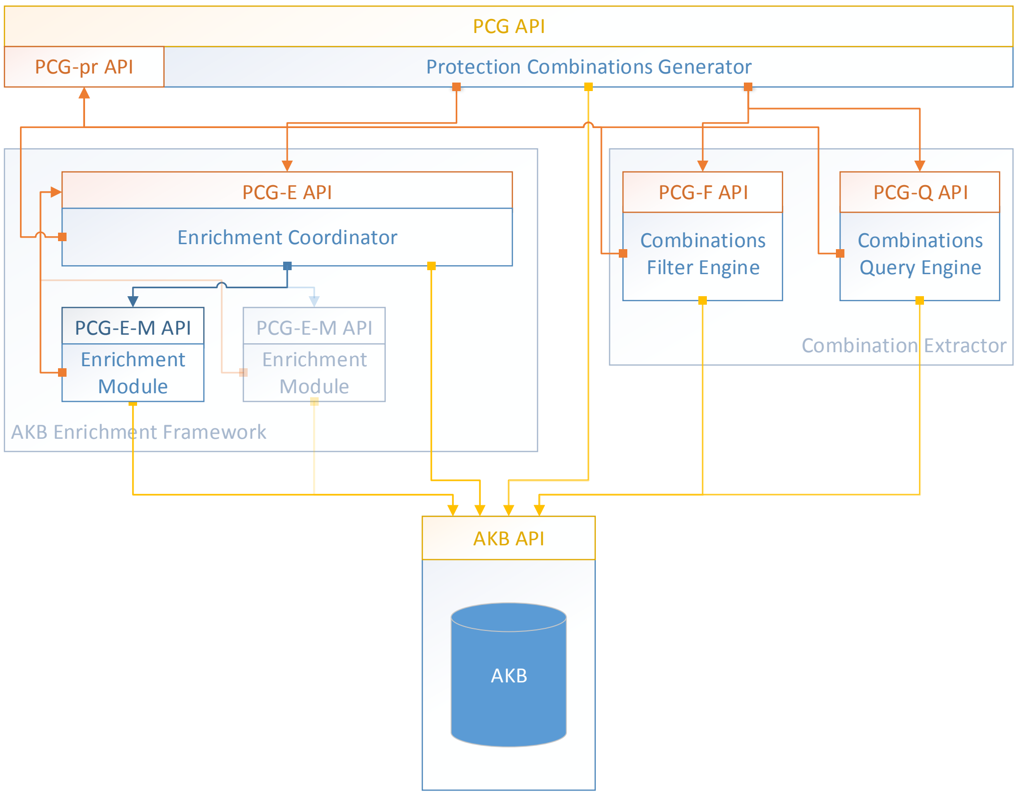

Figure 31: The architecture of the Protection Combination Generator.

(e.g., because of early pruning or because it processes user preferences). These generation limiting directive will be named filters.

This component also calls the ADSS `notifyTermination()` method when it completes its task.

PCG API

public API:
- `generate(asset)`: generate all the suitable combinations for a specific asset;
- `generateAll()`: generate all the suitable combinations for all the assets.

private API:
- `notifyTermination(moduleID)`: used to coordinate the internal component workflow, this method serves to notify the termination of a module with ID "moduleID".

PCG-EC Enrichment Coordinator API

- `runAll()`: executes all the EMs needed to enrich the AKB and generate the suitable combinations;
- `run(EM_ID)`: runs a specific EM identified with EM_ID;
- `getEMList()`: outputs the list of available EMs;
- `getExecutedEMList()`: outputs the list of EMs that have been executed during the enrichment;

PCG-EM Enrichment Module API

- `run()`: executes its ontology enrichment task and deletes all previously inferred data;
- `getInferredData()`: outputs the inferred data;
- `commit()`: commits the inferred data to the AKB;

PCG-F module API

- `run()`: reads the AKB and generates filters;
- `getFilters()`: outputs the filters it generated;
- `commit()`: commits the filters into the AKB;

PCG-Q module API

- `generateCombinations(asset)`: generates the suitable combinations for the specified asset;
- `generateAllCombinations()`: generates all the suitable combinations to protect all the assets in the target application;

H.4 ADSS Pruning Module

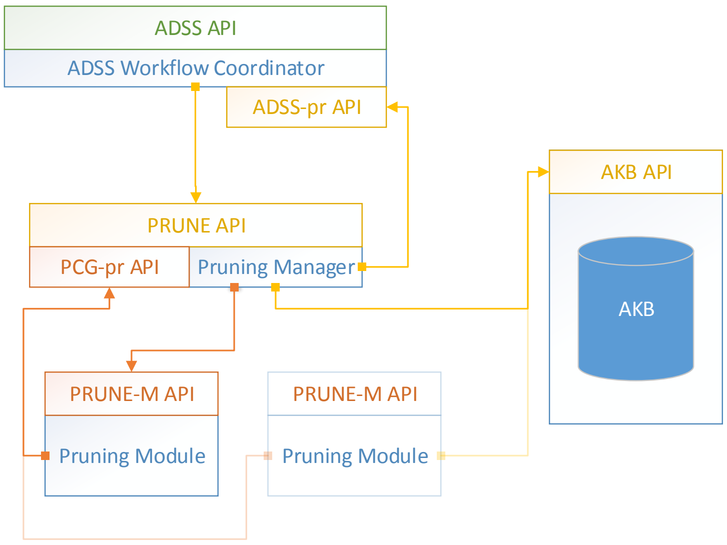

Figure 32: The architecture of the ADSS Pruning Module.

Description of the component:

This component performs the Pruning phase (see Figure 32). Different pruning strategies can be implemented by different Pruning Modules. One or more Pruning Modules can be executed during the Pruning phase. Each Pruning Module will be provided with a Pruning Module Manifest File which is used by the Manager to determine how and when to call it.

This component also calls the ADSS `notifyTermination()` method when it completes its task.

PRUNE API:

public API:

- `setContext(contextParams)`: sets the context passing the required parameters to perform the pruning;
- `run()`: based on the context information decides which combination to prune;

private API:

- notifyTermination(): used to coordinate the internal component workflow, this method serves to notify the termination of a module with ID "moduleID"

PRUNE-M API:

- `setContext(contextParams)`: sets the context passing the module the required information to perform the task;
- `getPrunedSuitableCombinations()`: outputs the pruned suitable combinations;
- `run()`: based on the context information, prunes suitable combinations;
- `commit()`: commits into the AKB information to mark as pruned the selected suitable combinations.

H.5 ADSS Optimization Module

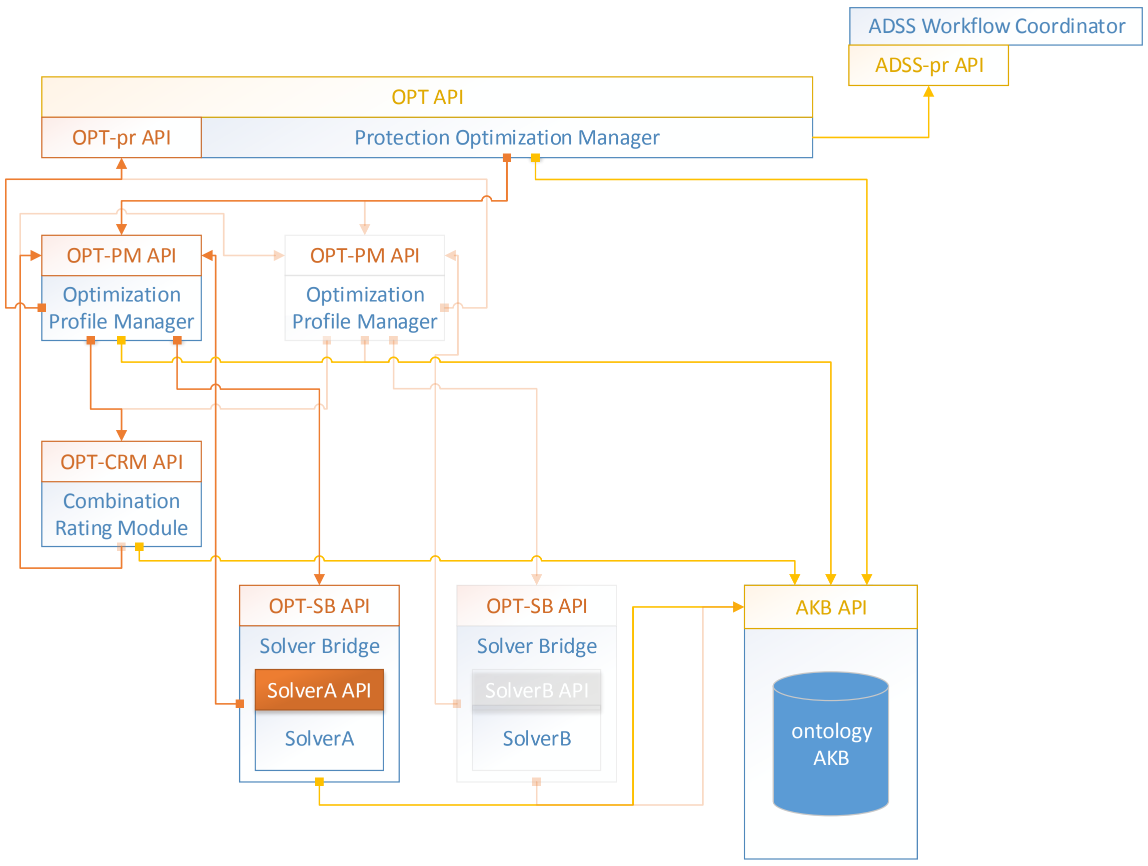

Figure 33: The architecture of the ADSS Optimization Module.

Description of the component:

This component performs the Optimization phase (see Figure 33). It is composed of several Optimization Profile Managers which are in charge to generate the optimization program and pass it to the proper Solver Bridge components. A Solver Bridge wraps a solver and passes it to the optimization model received by the Optimization Profile Manager. Each Solver Bridge is provided with a Solver Bridge Manifest File which is used by the Protection Optimization Manager to determine how and when to call it. Each Optimization Profile Manager is provided with an Optimization Profile Manager Manifest File which is used by the Protection Optimization Manager to determine how and when to call it. The Protection Optimization Manager coordinates this phases and it is in charge for selecting the proposer Optimization Profile Manager. Additionally, each Optimization Profile Manager is able to contact the Combination Rating Module, which is in change to perform the Evaluation phase. The Optimization Profile Manager asks Combination Rating Module to rate the non-pruned suitable combinations according to the rating categories, that is, the measurable features it uses in its target function.

This component also calls the ADSS `notifyTermination()` method when it completes its task.

OPT API:

public API:
- `setContext(ContextParams)`: sets the context passing the required parameters to generate the optimization model;
- `run()`: based on the context information, this method calls the Protection Optimization Manager, which calls proper Optimization Profile Manager;
- `commit()`: commits the golden combination into the AKB;
- `getGoldenCombination()`: outputs the computed golden combination;

private API:
- `notifyTermination(moduleID)`: used to coordinate the internal component workflow, this method serves to notify the termination of a module with ID "moduleID".

Optimization Profile Manager (OPT-PM) API:

- `buildOptProgram(cominationsSpace, optimizationProfile)`: build the optimization program;
- `commit()`: commits the optimization program into the AKB;

Optimization Combination Rating Module (OPT-CRM) API:

- `rateCombinations(measurableFeature)`: rates suitable combinations according to a given rating category;
- `rateCombinations(combination, measurableFeature)`: rates one suitable combination according to a given rating category;
- `rateCombination(combination)`: rates one suitable combination according to all the rating categories;
- `rateCombinations()`: rates all the suitable combinations according to all the rating categories

Optimization Solver Bridge (OPT-SB) API:

- `solveOptimizationProgram()`: solves the model using its internal solver.

H.6 Tool Flow Bridge

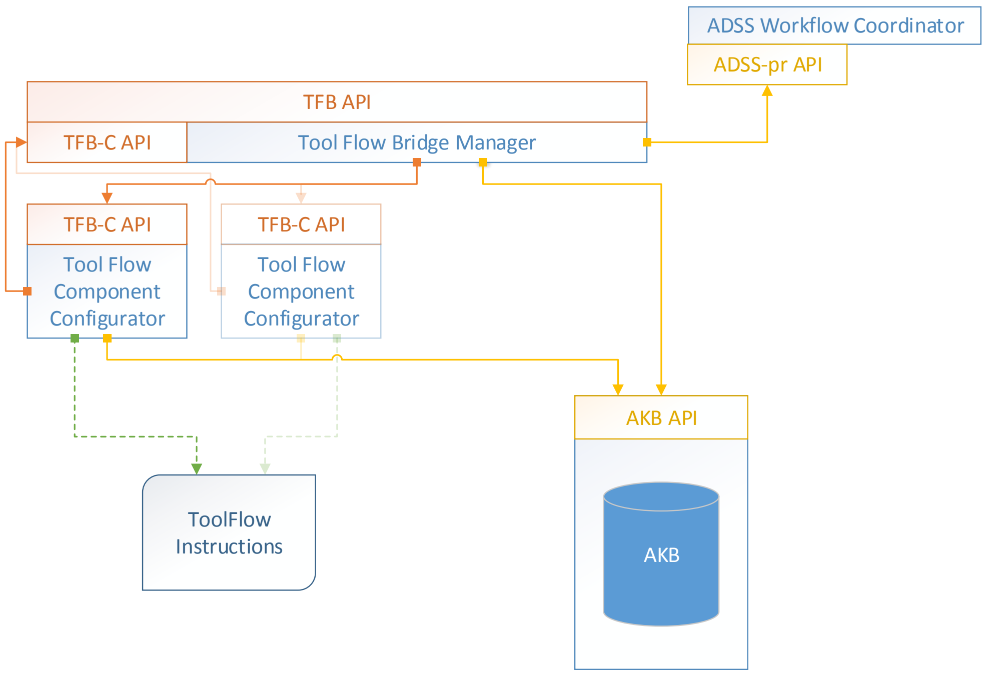

Figure 34: The architecture of the Tool Flow Bridge.

Description of the component:

This component performs the Tool Flow Instruction phase (see Figure 34). It is composed of Tool Flow Bridge Manager that reads the golden configuration and calls the proper Tool Flow Component Configurator. There will be one Tool Flow Component Configurator for each of the protection techniques that can be used in the tool flow (or protection techniques available at the user tool flow). Each Tool Flow Component Configurator will be provided with a Tool Flow Component Configurator Manifest File which is used by the Tool Flow Bridge Manager to determine how and when to call it.

This component also calls the ADSS `notifyTermination()` method when it completes its task.

TFB API:

public API:
- `getToolFlowInstructions(combination)`: outputs the instructions for all the ASPIRE tools needed to enforce the `combination)`;
- `getToolFlowInstructions(combination, tool)`: instruct a specific ASPIRE tool to enforce a protection combination;

private API:
- `notifyTermination(moduleID)`: used to coordinate the internal component workflow, this method serves to notify the termination of a module with ID "moduleID".

TFB-C API:

- `getToolConfiguration(combination)`: by reading the golden configuration, generates the configuration files/scripts needed for a specific ASPIRE tool and output it into an external Tool Flow Instruction File
- `commit()`: saves the Tool Flow component instructions into the AKB.

List of abbreviations

Abbreviation	Meaning
ACTC	ASPIRE Compiler Tool Chain
ADSS	ASPIRE Decision Support System
AES	Advanced Encryption Standard
AKB	ASPIRE Knowledge Base
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
ASM	Attack Simulation Model
API	Application Program Interface
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
CFG	Control Flow Graph
DES	Data Encryption Standard
DES3	Triple DES
DL	Description Logics
DoW	Description of Work
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
ECB	Electronic Code Book
EBNF	Extended Backus–NaurForm
EM	Enrichment Module
GCC	GNU C Compiler
HTTP	HyperText Transfer Protocol
ILP	Integer Linear Program
JSON	JavaScript Object Notation
LP	Linear Program
MATE	Man-at-the-end
MITM	Man-in-the-middle
MOVW	Move wide (immediate)

Continued on next page

– Continued from previous page

Abbreviation	**Meaning**
MOVT	Move top
Mx	Month x. A reference to a specific time in the ASPIRE project. It refers to the x'th month since November 2013.
OWA	Open World Assumption
OWL	Web Ontology Language
OWL DL	Web Ontology Language, DL sublanguage
RA	Remote Attestation
RNC	Residue Number Coding
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
STB	Software Time Bomb
VFP	Vector Floating Point
VM	Virtual Machine
WB	White-Box
WBT	White Box Tool

References

[1] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

[2] ASPIRE Consortium. D1.02 - aspire attack model.

[3] Stefan Fenz, Andreas Ekelhart, and Thomas Neubauer. Ontology-based decision support for information security risk management. In *International Conference on Systems, 2009. ICONS 2009.*, pages 80–85. IEEE Computer Society, 3 2009.

[4] Stefan Fenz, Thomas Neubauer, Rafael Accorsi, and Thomas Koslowski. Forisk: Formalizing information security risk and compliance management. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–4, 2013. Vortrag: 2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W), Budapest; 2013-06-24.

[5] Free Software Foundation. Gnu compilers documentation, section 6.3 - attribute syntax, 1988-2014.

[6] Free Software Foundation. Gnu compilers documentation, section 7 - pragmas, 1988-2014.

[7] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–. IEEE Computer Society, 2004.