Advanced Software Protection:
Integration, Research and Exploitation

D4.01
Preliminary ASPIRE Security Model

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	1st November 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10
Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D4.01 / 1.0
WP and tasks contributing:	WP 4 / Task 4.1
Due date:	April 2014 - M6
Actual submission date:	16 May 2014
Responsible Organization:	UEL
Editor:	Paolo Falcarin
Dissemination Level:	Public
Revision:	1.0

Abstract:
This deliverable presents the initial design and the main concepts of the ASPIRE Security Model. It will serve as the core of the ASPIRE Decision Support System, and its two principal concepts are a knowledge base and simulation models that are respectively used to formally represent and evaluate the different MATE attacks that ASPIRE aims to mitigate. The security evaluation with Petri nets as simulation models is described by means of an example taken from the concrete attacks described in D1.02 ASPIRE Attack Model. A snapshot of the design of the knowledge base is presented, i.e., of its preliminary design in this stage of the project, and background information is provided on security modelling in general, and on the use of ontologies and OWL-DL to implement the knowledge base.

Keywords: Security Model, Petri Nets, Knowledge base, Ontology, OWL.

Editor

Bjorn De Sutter (UGent)

Paolo Falcarin (UEL)

Contributors (ordered according to beneficiary numbers)

Cataldo Basile, Daniele Canavese (POLITO)

Mariano Ceccato (FBK)

Shareeful Islam (UEL)

The ASPIRE Consortium consists of:

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

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

Executive Summary

This deliverable presents the initial design and concepts of the ASPIRE security model, i.e., the integrated set of modelling techniques that will be used to formally represent and evaluate the different MATE attacks on software assets (within binary code executed on mobile devices). The security model represents the core of the overall ASPIRE approach to model and evaluate attacks, as well as the main technologies used in this approach and in the ASPIRE Decision Support System (ADSS), which will store a priori knowledge as well as application-specific information; the former will be added by ASPIRE security experts, while the latter will be derived from the application code, by means of code metrics, source code annotations, and empirical experiments aiming at getting actual data about attack times and success rates on an application.

The security model defines the simulation models (Section 2) and the structure of the knowledge base (Section 3), which are the basis of the ASPIRE protection evaluation methodology that will be further developed and extended in WP4, and of the ADSS that will be developed in WP5.

We have chosen Petri net models to represent attacks as simulatable models. They can be used to populate the knowledge base and to compute the expected attack times and attack success probability. We will compare different protections by comparing their effects on the times and success rates computed on Petri nets automatically generated for the protected applications. The Security evaluation with Petri nets is described through the One-Time-Password generator example that was described informally in D1.02 ASPIRE Attack Model.

We have started designing the knowledge base, using a set of UML class diagrams to represent the main concepts and their relationships, and we have chosen ontologies to formalize all the concepts in such diagrams into description logic, in order to provide a precise relation between assets, attacks, and protection techniques. The presented diagrams of the main model at the core of the knowledge base and of the sub-models that refine the main model, are just a snapshot of the current design. Refining the knowledge base to include as much as possible the relevant and useful information will be an iterative process that will span most of the project.

Finally, this deliverable includes some background on ontologies and a comparison with related works is presented.

Contents

List of Figures

Section 1 Introduction

Chapter Authors:

Bjorn De Sutter (UGent)

1.1 Role of this deliverable

One of the ASPIRE goals is to provide measurable protection against MATE attacks. The scope of the attacks to be considered was already presented in D1.02. That report also provided an informal overview of the attack techniques and attack tools to be considered, of their robustness against existing protections, of their identification and exploitation metrics, and several examples of how different attack (steps) are combined in the real-world to attack actual applications and their assets.

In the context of the economic model of MATE attacks that was also discussed in D1.02, measurable means that the impact of applied protections on an attacker's effort for engineering an attack and on his exploitation potential can be quantified. In terms of Figure 1, which is repeated here from the ASPIRE DoW and from D1.02 for the sake of self-containment, we want to quantify the changes in the blue surface area and in the red surface area that result from applying a set of protections.

Furthermore, we want to be able to quantify these effects by means of fully automated tools. Of course the developer has to provide some of the necessary information, such as the application and a description of the assets to be protected, the envisioned threat (at a very high level of abstraction), and perhaps information on the targeted platform. But apart from that, ideally the developer should not have to perform manual computations or analyses. Instead, with a fully automated toolbox, the developer can ask the tools to compute the optimal combination of protections for his software.

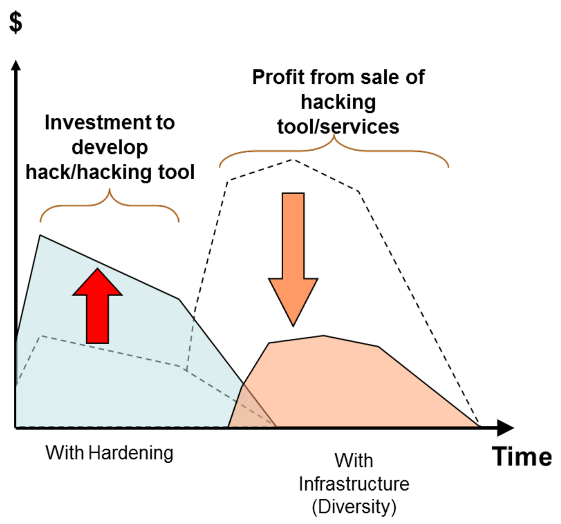

Figure 1: MATE attack economics

Developing a toolbox that requires only the minimal input from a developer to protect his application and its assets obviously requires that the tool box comprises a large, generic knowledge base, in which all relevant information from D1.02 and other sources is present. Furthermore, the toolbox will need to be able to instantiate its general knowledge for the developer's case at hand. In other words, from the general knowledge contained in the knowledge base, and from the developer's input, the toolbox will need to synthesize a model of the developer's application in relation to the threat he described at a high level of abstraction. Then on the basis of that synthesized model, the toolbox has to be able to reason about the impact that the supported protections can have on the threat to be mitigated. Ultimately, the toolbox can select the most appropriate protections based on the models. The toolbox can report the selected protections and visualize their impact on envisioned attacks for validation by the developer, and the toolbox can generate a configuration for the protection tool chain to actually apply the protections.

In the remainder of this document, we will use the term ASPIRE Decision Support System (ADSS) to denote this toolbox. Its development is a major goal of the ASPIRE project.

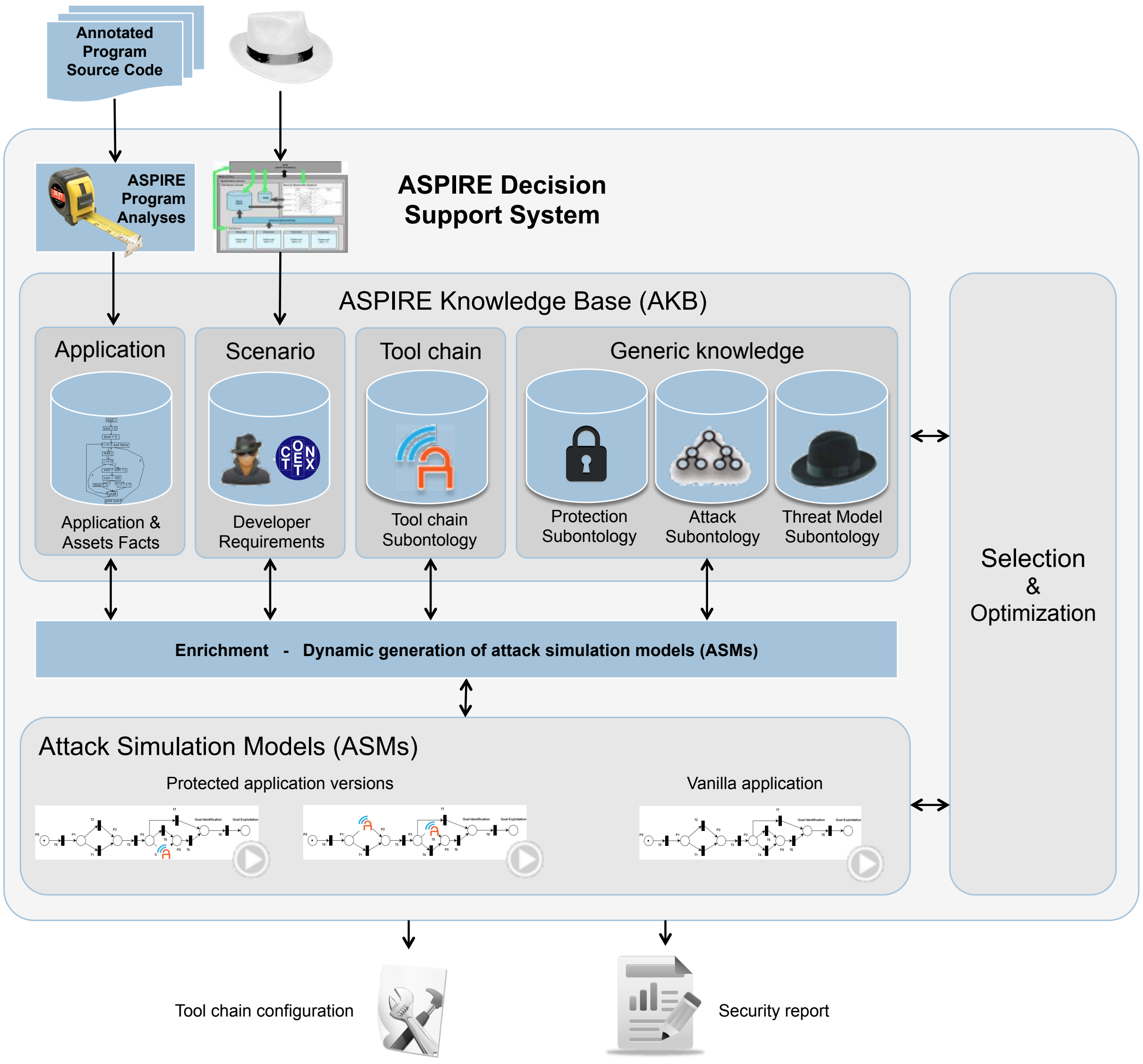

Figure 2: The ASPIRE security modelling approach

The role of this deliverable then is to present the overall ASPIRE approach to model and evaluate attacks, as well as the main technologies used in this approach and in the ADSS. Two central pieces of the envisioned approach are concrete **simulation models** of attack/protection combinations, and a **knowledge base**. The simulation models will allow us to quantify the impact achieved by some protections. The knowledge base will store all information necessary to generate the concrete simulation models given the input from a programmer. In essence, the knowledge base provides a precise relation between assets, attacks, and protection techniques. To assemble this knowledge, we will rely on the findings described in D1.02 Attack Model (e.g., the attack overview and attack paths).

This deliverable defines the simulation models and the structure of the knowledge base. As such, it defines the basis of the ASPIRE protection evaluation methodology that will be further developed and extended in WP4, and of the ADSS that will be developed in WP5.

1.2 Overview of the ASPIRE Security Modelling Approach

Figure 2 presents a conceptual overview of approach that will be developed in ASPIRE. As shown on the top left of the figure, the ADSS will take as first input the application to be protected. The developer is expected to have annotated the source code to mark its assets to be protected: code fragments, variables, keys, etc.

Secondly, the developer (shown as the white hat) will also provide additional information to the ADSS, such as 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, business model, assumed online connectivity, server requirements, etc.

The ADSS will then first analyse the application (using the source code analysis and compiler tools from the ASPIRE tool chain) to identify the assets as indicated by the programmer's annotations, and to extract all relevant information about the application. Besides information describing the assets to be protected, this information includes, but is not limited to, profile information, code size, slices, points-to information, etc. All the thus collected formal application and asset facts will be represented in data structures that facilitate reasoning about the application, and be stored persistently such that they can be reused later on, if necessary, without wasting resources on re-computing already computed information.

The AKB will also model all additional information provided by the developer to the ADSS. In general, this information defines the requirements that need to be met by the protections in the developer's scenario. Combined, the requirements and the application and asset facts form all the application/execution specific knowledge.

Besides that specific information provided by the developer, the AKB also features a priori knowledge that is valid for all applications of the ADSS. That a priori knowledge comprises generic knowledge on the one hand and tool chain knowledge on the other hand.

The generic knowledge captures all information that is generally applicable and relevant in the context of software protection against MATE attacks. Fundamentally, it formalizes the relationships between assets, threats, attacks, protections, and attack attributes as we described informally in D1.02. Some examples of knowledge that needs to be modelled formally, and for which other components in the ADSS will need to be able to query the AKB, are the following:

- Given some type of asset and a threat (see D1.02 Section 3), what are the attacks paths that an attacker can use?
- Which attack steps require an internet connection?
- Which protection techniques are incompatible/compatible with a particular compiler/OS/CPU?
- For each attack step, what are the relevant metrics (e.g., control flow complexity or size) that determine the effort required by an attacker of a given level of expertise?
- What is the best-estimated relation between that effort and the metric (linear, quadratic,...) ?
- For each attack step, which level of expertise is needed to deploy it?
- For each attack step, metric and type of asset, what is the relevant part of the application (e.g., a function, a module, ...) on which the metric has to be computed?
- For each protection, how does it influence the relevant metrics?
- Which protections can be combined in a given code fragment, and which not?

In principle, this generic knowledge is independent of the tool chain that will deploy the protections. For that reason, it is also necessary to model the capabilities of the tool chain, i.e., the available protections, the combinations of protections supported by the tool chain, the granularity at which protections can be deployed (e.g., at the function level, or on sets of basic blocks within functions, or on sets of basic blocks spanning multiple functions, etc.), and the options and parameters that need to be specified for each protection to be invoked.

When all of the discussed information is merged in the AKB, it will model all necessary information to reason about the protection of the application at hand, given the requirements of the developer, the capabilities of the tool chain, and all available knowledge about the links between protections, attacks, assets and threats. Simply merging all information into one database will not suffice to reason efficiently and effectively, however.

We will therefore externalize some of the reasoning to external modules that interact with each other to enrich the AKB, i.e., to derive new relationships to be added to the AKB on the basis of the already existing ones. For example, when it is known that a certain code fragment needs to be protected through obfuscation, when the execution frequency of the code fragment is known, as well as the per execution overhead of some obfuscating transformations then the ontology can be enriched with the overhead that each obfuscation will introduce if applied on this code fragment.

Once the enriched AKB incorporates all relevant static knowledge, the selection & optimization module will start querying it for possible interesting protections. On the basis of the retrieved information, the ADSS will generate multiple dynamic attack simulation models (ASMs). Each ASM will concretely model the possible attack paths on a protected version of the application, i.e., on a version to which a specific combination and configuration of protections has been applied.

For the ASMs, we have chosen to use Petri nets, in which the transitions correspond to attack steps. Each transition will be annotated with probabilistic models of the expected time needed to perform the attack step, as well as with probabilistic models of the expected success rate of the attack step.

Using existing simulation tools, the ADSS will then simulate each instantiated Petri nets. With this simulation, these tools compute the distribution of the time needed to perform the whole attack and the overall likelihood that the whole attack succeeds on the modelled application version. After each simulation, the ADSS will be able to enrich the AKB with the computed results. So at that point, the AKB is enriched with information on the strength of different protections for the concrete application, scenario, and requirements at hand.

During the selection and optimization process, many different ASMs will be generated and simulated. When the AKB is enriched sufficiently with results of Petri Net simulations, the ADSS can compare the resistance of the different simulated combinations of protections against the envisioned attacks, and select the best combination and their configuration.

This will ultimately allow the ADSS to generate a configuration for the ASPIRE tool chain, which will then apply the selected protections on the application at hand.

Furthermore, a security report will be produced on the findings of the ADSS that allows the developer to validate that appropriate protections have been selected. Ideally, that report would not be a static enumeration of human readable facts, but a computer-readable, structured representation of all relevant data that can be browsed and examined though a specialized GUI.

In practice, however, we will probably lack the resources to engineer such a GUI, so we will likely have to do with a research version, and plan the development of such a GUI for the exploitation of the ASPIRE results. The reason is simply that enough major research challenges remain to be tackled within the ASPIRE project to make this proposed security modelling approach work, i.e., to have it make good selections of protections to be applied.

1.3 Challenges to Address

Some of those research challenges are the following:

1. **How to avoid combinatorial explosion in the generation of ASMs?** In general, the question to be answered is which and how many ASMs we need to generate and simulate to let the ADSS make good decisions. Given the large number of protections and parameters we envision for them and the large set of possible code fragments to apply them to, it will definitely be needed to prune the search space at some point. This will mainly be studied in the remainder of Task 4.1 of WP4.
2. **How to quantify the required effort/time and success probabilities of the individual attack steps?** Useful metrics that relate effort/time and success probability to code properties will be researched in Task 4.2 of WP4, and human experiments for obtaining concrete relations will be set up and executed in Task 4.3 of WP4.
3. **How to model the mutual protection that different protection techniques provide?** In some cases, the increase in attacker effort resulting from two protections combined will be greater than the sum of the increases in effort of the individual protections. How to incorporate that knowledge is an open question, which the project will try to answer when deploying the developed infrastructure on the use cases.
4. **How much source code annotation and user guidance (i.e., input) will the ADSS need?** Using heuristics and compiler analyses, the ADSS needs to be able to determine exactly which protections are relevant, and to which code fragments they might need to be applied. However, data flow and control flow analyses in compilers have limited precision because of decidability issues. To what extent the limited precision needs to be overcome by additional annotations and user guidance is an open question at this point. The project will also try to provide an answer by experimenting with the developed infrastructure on the use cases.

These are just a few of the many research challenges ahead of us. So at this stage of the project, we can definitely not answer all questions that the proposed security modelling approach might raise.

We are confident, however, that this proposed approach is the best available approach for security modelling in the domain of MATE attacks in general, and for the ASPIRE project in particular. While this overall approach is, to the best of our knowledge, new in the domain of software protection, it has been researched and deployed in different security contexts, such as the MOSES [Mos13, Mos14] and PoSecCO [Pos31, Pos32] FP7 projects. Furthermore, we can rely on existing technologies and tools that fit our purpose, as will be explained in this deliverable and in some of the follow-up deliverables, and the ASPIRE consortium partners have extensive experience with the underlying techniques and tools.

1.4 Structure of the Deliverable

Section 2 describes the methodology for security evaluation through Petri nets, which are used to model and evaluate attacks against the software assets the ASPIRE technology is supposed to protect.

Section 3 presents the ASPIRE knowledge base which contains the high-level definition of the ontology and its data structures to represent the concepts need to model assets, attacks and protections, from which to generate appropriate Petri nets.

Section 4 introduces the background on ontology engineering and discusses the related works is security modelling.

Section 5 concludes the document by describing the plan for the security model and his enhancements, and the connections with the other tasks.

Section 2 Security Evaluation: Petri Nets

Chapter Authors:

Paolo Falcarin (UEL), Bjorn De Sutter (UGent)

2.1 Introduction

As described in the introduction, the ASPIRE security modelling approach relies on the simulation of formal models that mimic the activities of actual attackers in terms of the economic model of MATE attacks. Concretely, we will use models that allow us to draw conclusions on the time/effort required by the attacker and on his chance of success, given all the attack steps he needs to perform.

In Section 5 of D1.02, several attack paths were discussed on different types of applications and different types of assets. In essence, each attack path is a sequence of attack steps or attack tasks with which consecutive sub-goals of the attacker are reached until his ultimate goal is reached.

Modelling attacks as pure sequences of attack steps would not be realistic, however, because in practice diverse attack paths might lead to the same goal for the attacker. For example, on unprotected applications, an attacker might use static structural matching techniques, or dynamic matching techniques, or sniffing to find the location where a secret key is computed or used within a program.

So instead of using sequences or even sets of sequences to model all possible attacks corresponding to a threat on an asset, researchers have proposed to model attacks by attack trees that combine all the attack paths leading to a specific attack goal [Sch99]. Each single attack tree has a root node corresponding to the ultimate goal of the attacker. In the tree, each parent node's child nodes model the sub-goals (and corresponding attack steps) necessary to meet the parent node's goal. Nodes can be AND nodes or OR nodes. AND nodes denote that all sub-goals need to be reached in order to meet parent's goal. OR nodes denote any of the child's sub-goals suffice to reach the parent's goal. In the example of Figure 3, the root of the tree is the main attack goal, and it is an AND node as its child-nodes are sub-goals SubGoal1 and Subgoal2 which both have to be achieved. Attack1 or Attack2 can be used to achieve SubGoal1, while Attack3 has to be used to achieve SubGoal2. This way, attack-tree models are not exactly trees as defined in graph theory, but they are actually AND-OR graphs.

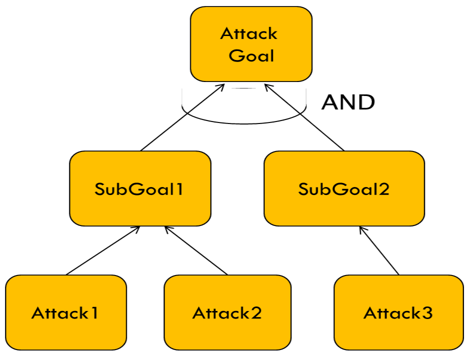

Figure 3: An example attack tree in the form of an AND-OR graph

Such graphs still suffer from the problem that they cannot model the fact that in practice, one attack step, such as undoing all obfuscations of a certain type in an application, can let the attacker meet several sub-goals at once. In other words, the tree structure of the AND-OR graphs does not suffice to model realistic attacks.

Attack graphs have therefore also been proposed to overcome the limitations of attack trees [Gup07,She03,Ou06,San13]. The different proposals of attack graphs share the representation of attack goal as nodes and attack tasks as transitions and their behaviour is similar to state-transition diagrams, even if each proposed approach uses a slightly different semantics in their attack graph representation. State-transition diagrams cannot model concurrency, however, as needed to model two or more attackers working as a team.

In this project, we will therefore instead rely on Petri nets [Pet66], which are a super-set of state-transition diagrams, allow to model concurrency, and have more consolidated mathematical foundations. In fact a state-transition diagram can be seen as a Petri net where every transition has only one incoming arc and only one outgoing arc. And like state-transition diagrams, Petri nets can also have non-determinism when more than one transition can be enabled from one place.

Petri nets are bipartite graphs, i.e., every node of one type is adjacent only to nodes of the other type. In Petri nets, these two types of nodes are called places and transitions. In the context of the ASPIRE security model, the nodes have the following function:

1. **Transition nodes model attack steps (i.e., attack tasks).** These nodes model activities performed by the attacker (e.g., disassembling, static analysis, debugging, etc.) and have several attributes, such as distributions of the required effort/time and probability of success as a function of the considered features of the code under attack, as well as some function that describes the relevant features of input for the next attack step in terms of this attack step's input.
2. **Place nodes model attack states (or attack goals),** which are reached following one or more attack steps (e.g., identification of a secret, disabling a protection).

An attack model based on Petri Nets was first proposed by McDermott [10] and later Wang et al [Wan12]. The innovation of our security evaluation approach is that we are not only focused on modelling the attacker's viewpoint with a Petri net, but that we also aim for simulating the attacker's behaviour before and after protections have been applied in order to evaluate protections efficacy in terms of introduced delay and reduced attack success probability.

In our case, the simulation will involve relevant data related to attacker effort and success probability. This information will be retrieved from the AKB, which will store all information collected from code metrics (Task 4.2) and by doing manual attack experiments (Task 4.3 and Task 4.4 in WP4). The simulation of the Petri Nets as a model of the attacker will be performed using free external tools like CPN-Tools or PIPE [CPN, PIP].

By relying on code metrics related to practically relevant code features and by obtaining data from manual attack experiments (as some others have also done in the past [Sut06,Lin03,Udu05,Ham11,Zha12]), we want to overcome the limitations of theoretical analyses of protection strength [Col97,Myl05,Dal05,Wan00]. The latter are useful in theory, but except for very limited attack scenarios, they have not been validated in practice.

In the remainder of this section, we present a brief overview on the fundamentals of Petri Nets, followed by an example of modelling the One-Time-Password attack path (as discussed in D1.02 Section 5.3) with a Petri net, and a discussion on how to evaluate the attack cost of such an attack in terms of time and success probability.

2.2 Overview of Petri Nets

Petri Nets were first introduced by Carl Adam Petri to model sequences of chemical reactions [Pet66]. Later they have been utilized in many different computer science fields, for example to model concurrency and synchronization in distributed systems [Pet77].

Petri nets are similar to State Transition Diagrams as they use a visual representation to model the system behaviour, and they are based on strong mathematical foundations. A Petri net consists of three types of components: places (circles), transitions (rectangles) and arcs (arrows):

- Places represent possible states of the system;
- Transitions are events or actions which cause the change of state;
- Every arc connects a place with a transition or a transition with a place.

From a graph theory viewpoint, a Petri net is equivalent to a bipartite directed graph where places and transitions represent two types of nodes. A bipartite graph is a graph where every node is only adjacent to nodes of the other type. In a Petri Net an arc can only connect a place with one or more transitions and a transition with one or more place). The places with outgoing arcs leading to a transition are called the input places of the transition; the places with incoming arcs (outgoing from a transition) are called the output places of the transition.

A change of state is denoted by the transfer of a token (represented with a black dot) from place to place and is caused by the firing of a transition. The firing represents an occurrence of an event or an action taken; the firing is subject to token availability: in the initial state of the Petri net (Fig.1a) there is a token in place P0 and when the transition T0 fires the token is moved from its input place P0 to its output place P1 (Fig. 1b). Chaining different places and transitions will produce a net representing a sequential execution of transitions firing one after the other.

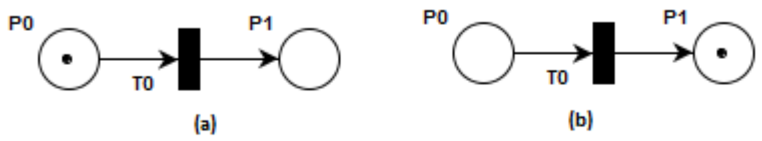

Figure 4: A simple Petri net in its initial state (a) and after the transition fired (b)

In the context of ASPIRE, we will use Petri nets as a form of attack graphs. Places will represent attack sub-goals, such as the fact that some information was acquired (e.g., a key was stolen or a relevant piece of code was identified) or that some task can be accomplished (e.g., a key check was removed from an application, a debugger attached, or a trace collected, etc.). Transitions then correspond to the attack tasks necessary for the attacker to proceed from one sub-goal to the other. The start place denotes the start of the attack, and the set of sink place denotes the ultimate goals of the attack that have been reached..

In general, a Petri net transition is fireable or enabled when there is at least one token in each input place (Figure 4a); after firing, tokens will be transferred from the input places (old state) to all the output places, denoting the new state (see Figure 4b). When a transition T fires, one token per input place is consumed and one token is created in each output place. So in Figure 5a, transition T can fire if and only if both P0 and P1 have at least one token each; in this case transition T acts as a synchronization point waiting for both token to be present at the input places (P0 and P1) in order to proceed with the concurrent execution (Figure 5b) of three paths starting from the output places P2, P3, and P4.

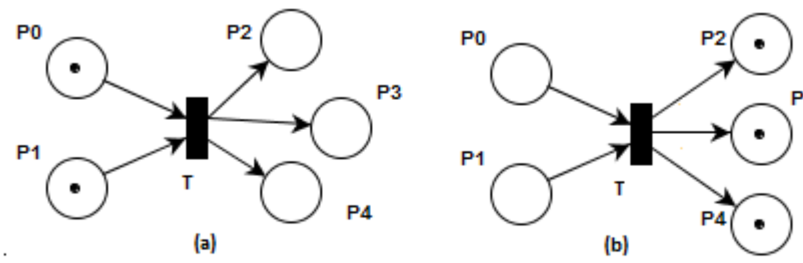

Figure 5: Synchronization (a) and Concurrency (b)

In the context of ASPIRE, Figure 5a can represent the behaviour of an AND node in an AND/OR graph. It models the case of an attack step T requiring to wait for two types of information (from P0 and P1) to perform T and then proceed from there with three possible independent paths of attack execution. For example, P0 could be the result of a previous attack step consisting of static analysis of binary code to detect key-processing code, while P1 could be the result of another previous attack step consisting of static analysis to detect code portions writing into the memory: both information are needed to proceed with the next attack step of dynamic analysis to detect the value of the decrypted key in memory.

The OR behaviour, which is needed in ASPIRE to model alternative attack paths leading to the same (sub-)goal, can be presented as in the example in Figure 6a: in this case either T1 or T2 can fire and move their token to P3; in case T1 fires, the net will move to the configuration of Figure 6b. P1 can be the state in which a static control flow graph of the program code have been reconstructed, and P2 the state in which a program trace has been collected on some inputs. Then in order to identify the location of some code, static structural matching techniques can be used on the control flow graphs (T1), and dynamic matching techniques can be used on the trace (T2). Either of T1 and T2 can lead to sub-goal P3, which is the identification of, e.g., the AES code in the application under attack.

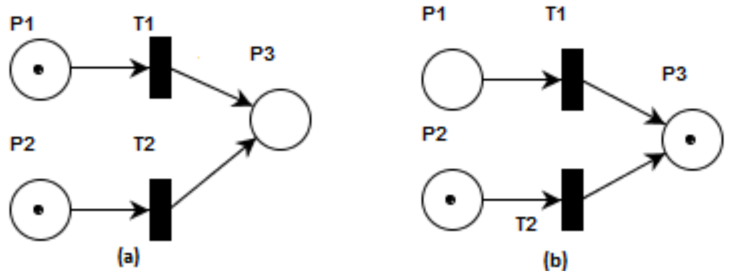

Figure 6: Place P3 is reached by firing either transition T1 or transition T2

Figure 7 shows another pattern recurring in Petri nets: choice. The choice pattern is somewhat equivalent to an if-then-else construct in a programming language where only one of the two branches can be executed: in other words, if T0 fires first then the token is moved to P1 through T0, while if T1 fires first the token will move to P1 through T1. In the ASPIRE attack modelling context, choice indicates that an attacker can choose which attack technique to deploy to reach a certain (sub)goal, and that in the worst case from the defender's perspective, he will choose the fastest one. Alternatively, it also models that when multiple attackers cooperate and try alternative techniques concurrently, they will be able to proceed as soon as one of them is successful. In other words, this model of choice assumes the best-case scenario for attackers, which corresponds to the worst-case scenario for defenders. As such, the use of this model will not lead to underestimations of the effort an attacker might need.

Graphically, places in a Petri net may contain a discrete number of marks called tokens. In general tokens can represent pieces of information flowing through the net. Any distribution of tokens over the places will representing a configuration of the net is called a "marking", which is the global state of the Petri net. Each transition can update the information flowing through the net with the token.

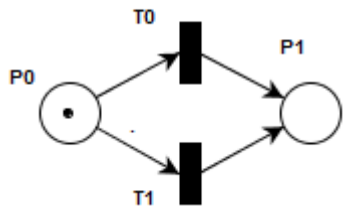

Figure 7: Choice in a Petri net

In general, the execution of Petri nets is nondeterministic: when multiple transitions are enabled at the same time, any one of them may fire. Since firing is nondeterministic, and multiple tokens may be present anywhere in the net (even in the same place), Petri nets are usually well suited for modelling concurrent systems. In the context of ASPIRE, we will reuse that to model attacks with sequential as well as (potentially) concurrent attack steps.

When we will simulate a Petri net in ASPIRE (using existing tools), we will compute all the different ways in which a token can flow through the net, and we will aggregate all relevant information regarding effort, probability, and (sub)attack target properties as the tokens flow through the net. That way, we will be able to compute the relevant features of complete attacks, i.e., of complete realizations by an attacker of threats on an asset starting from scratch.

In general, there is no need to let a Petri net start with a single source node. Moreover, the AKB will likely contain information on multiple possible attack paths, with different starting points. This will particularly be the case for applications protected with multiple lines of defence that strengthen each other, as such applications will by definition not provide a clear attack entrance point for the attacker. The dynamic generation of ASMs will therefore be able to extract multiple attack paths from the AKB and combine them into a single Petri net with different starting points leading to the same attacker goal. During the simulation of such a Petri net, we can then simulate multiple attackers working in parallel by associating one token per attacker at different starting places (i.e., a marking composed of more than one token).

In literature, the high-level term Petri net is used for many Petri net formalisms that extend the basic Petri net formalism. This includes:

- Coloured Petri nets, where tokens are of different "colours", i.e., objects of different type [Jen09].
- Hierarchical Petri nets [Hub90], where each transition might be another embedded Petri net: in our case an attack step can be a general attack actually composed of a sub-net of other low-level attack steps.
- Timed Petri nets [Wan98], where each transition has a delay, representing the time needed to execute a transition.

In the ASPIRE Security Model, we will build on the timed Petri net extension to compute overall attack effort/time. Section 2.4 gives an example of how we plan to do so.

Hierarchical Petri nets will also be used to facilitate the merging of different models where one more abstract attack step can be further detailed by a sub-net containing more concrete attack steps and paths.

In the planned research in WP4, we will also study which other extensions might be applied to our ASPIRE Security model. For example, coloured Petri nets might be useful to represent, with tokens of different colours, the different types of data flowing through a net that are relevant to compute the time needed to perform each attack step (like code size, execution traces, etc.). The planned research will determine whether this is useful and needed or whether better alternatives are available.

2.3 Example Model of Attack on Vanilla OTP

In this section, we explain how to use Petri Nets to model MATE attacks. We do so by means of a real-world example taken from D1.02 ASPIRE Attack Model, Section 5.3: Attacks on one-time password (OTP) generators. For the sake of self-containment, we repeat the definition of the OTP generator and of the possible attacks in this section.

2.3.1 OTP Definition (from D1.02 Section 5.3.1)

A One-Time Password (OTP) is generated by an OTP application on the basis of a seed and a counter, i.e., two secret values shared by a client device and a server. As part of the OTP application provisioning, the seed is sent to the client in an encrypted form by the server when the application is initialized on a device for the first time. The counter is then initialized to a contractual value. At each later invocation of the OTP application, the counter value is incremented and a new OTP value is generated and returned to the user, which can then use that OTP, for example to login to his online banking website. The algorithm used to generate the OTPs is designed in such way that it is not possible to guess the next OTP value from the current one. The OTP generation processing is furthermore protected by a Personal Identification Number (PIN) control to prevent that unauthorized people with access to the device can generate an OTP when only the authorized user should be able to do so.

2.3.2 Informal Vanilla OTP Attack Paths Description (from D1.02 Section 5.3.2)

First an attacker can try to bypass the PIN protection control to allow anyone, incl. himself, to generate an OTP. To identify the check, almost all static and dynamic reverse-engineering methods discussed in D1.02 Section 4 can be used. To bypass the check once it is identified, static or dynamic anti-tampering techniques can be deployed, as discussed in D1.02 Sections 4.3.3 and 4.4.5. Similar techniques could be used not to bypass the check, but to inject code into the application to steal the PIN, i.e., to send it to a remote server controlled by the attacker.

A generated OTP is an asset by itself, but is not interesting for an attacker that wants to generalize an attack.

Other interesting assets for an attacker are the seed and the counter, which the attacker would like to send to a remote server. The attacker can track the seed either during the provisioning of the OTP application when it is received from the server, or later when the seed is used during the actual generation of OTPs.

To retrieve the seed from the server response during the provisioning, the transport key is calculated on the fly and the deciphering starts, for example by means of AES. This code can be identified using the techniques discussed in D1.02 Section 5.2. A debugger or other dynamic analysis techniques (see D1.02 Section 4.4.4) can be used after the last decryption round to get the value of the seed as clear text.

In order to engineer such an attack from scratch, the attacker will either have to go through numerous uninstall/install rounds of the application to trigger the server to respond to enough provisioning requests that he can study using the dynamic attack techniques of D1.02 Section 4.4, or he will have to set up his own server to provide enough fake responses. With the former approach, the attack will be very time-consuming and uncertain, in particular when the server keeps track of successive interrupted provisioning attempts and blocks the involved devices. With the latter approach, that cause of delay can be avoided. However, the attacker still will have to work around the checks that the OTP application typically performs to ensure that it only runs the provisioning functionality once.

It is therefore much more efficient for the attacker to pinpoint the binary code location of the OTP generation in an already provisioned application and attack that code, which will also give him both the seed and the counter values at the same time. To do so, the attacker can often take advantage of a typical OTP computation patterns involving consecutive xor operations. Again a static analysis is required to locate the relevant code fragments, but this analysis can be focused using a small number of dynamic debugging sessions to locate the part of the code activated right before returning the OTP to the user.

2.3.3 Petri Net Model of Vanilla OTP Attack Paths

For the sake of demonstration, Figure 8 represents a simplified Petri net to model an attacker going after the seed used in a vanilla OTP generator that has not yet been protected with ASPIRE protections.

While this was not explicitly stated in the above informal description of the attack paths, this Petri net relies on the insight that the PIN is needed to start the provisioning of the OTP generator and also every time an OTP is to be generated.[1]

Phase 1: stealing or bypassing the PIN check

P0 is the starting state: the attacker has white-box access to the application under attack, but has not performed any activity.

T0 is the first attack step for the attacker: to identify the code section dealing with PIN checking: to achieve this, the attacker can use almost all static and dynamic reverse-engineering methods. So in a more complete model, T0 could be replaced by a more complex Petri subnet that models all those different methods. However, in the ASPIRE approach, this more detailed modelling will not necessarily be needed to obtain meaningful results. Because of the enrichment strategies that will be used, the AKB can be enriched with aggregate results for the commonly occurring subnets, thus avoiding that Petri nets that model concrete attacks grow too big.

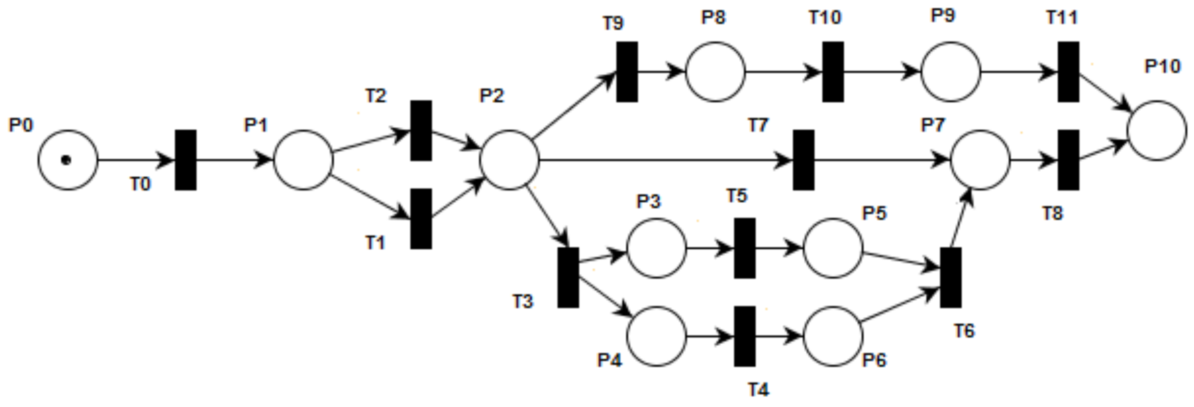

Figure 8: Petri net of the OTP attack

After reaching state **P1** through T0, **T1** and **T2** are alternative steps to bypass the check once it is identified: static or dynamic tampering techniques can be deployed to actually bypass the check in the code (**T1**) or to inject code into the application to steal the PIN (**T2**), i.e., to send it to a remote server controlled by the attacker.

At state **P2**, the attacker can execute the provisioning of the application as well as the OTP generation itself. As this is not really the asset he is after, he will continue along one of two paths: either retrieving the seed during the provisioning, or retrieving it during the OTP generation. The former is modelled in the subnet from P2 to T8 (going through T7 or through the other path starting with T3), the latter in the subnet from T9 to T11.

[1] This assumption is used for the sake of demonstration. In practice, a registration code instead of a PIN might be needed to start the provisioning.

2a: stealing the seed during provisioning

T4 models the attacker's tampering activities required to work around the built-in restriction in the application that its provisioning phase can only be executed once. That way, he does not need to reinstall the application every time he will want to observe the provisioning phase in action.

In parallel with the tampering of **T4**, in **T5**, the attacker or his accomplices can set up a fake server to provide provisioning answers during each run of that phase. If the attackers choose to steal the seed during provisioning, T4 and T5 attack steps can be executed in parallel (by two attackers) or they can be both executed in any order (if they are performed by one attacker only).

To represent the concurrent execution of this two attack steps, a dummy transition **T3** has been added before T4 and T5. This dummy T3 fires immediately after consuming the token from place P2, creating one token in P3 and another one in P4: at this point T4 and T5 are both enabled and they can fire independently; the following attack step T6 will be enabled when there will be a token in P5 and P6, meaning that both T4 and T5 have been performed: T6 will then consume both tokens from P5 and P6 and create a token in P7. This models that once the application has been tampered with (T4 performed, one token in P6) and the fake server is ready (T5 performed, another token in P5), the attacker can observe the execution of the provisioning phase dynamically in **T6** to locate the AES code that deciphers the seed received from the server. State **P7** models the state in which this code has been located.

Alternatively, and also to identify the AES code in the provisioning phase, in **T7** the attacker can try to perform dynamic analysis on an un-tampered provisioning phase that is uninstalled/installed between every two runs, and that actually connects to the real server in every run. This requires less preparation of the attacker but the constant re-installing will require more time.

When the AES code has then been identified in P7 through either of the attack paths, the attacker can use another dynamic analysis step **T8**, such as running the application with a debugger, to observe a seed as soon as it is received and decrypted during a new provisioning phase involving the real server.

Phase 2b: stealing the seed during OTP generation

For stealing the seed during the OTP generation, in step **T9**, the attacker will first observe several runs dynamically to isolate the code fragments executed right before the OTP is shown to the user on the display. That will prune his search space for the next step, **T10**, in which he uses static techniques to identify the chain of XOR operations involved in the OTP generation.

Once the xor operations have been identified, he can use another dynamic analysis in step **T11** (which is similar to T8), to observe the value of the seed as it is being consumed by the chain of xor operations.

2.4 Security Evaluation of the Vanilla OTP Generator

Once the Petri net structure that models the possible attack paths has been identified, we need to evaluate the cost of those paths in terms of attacker effort, and the probability that the attacks actually succeed.

Each individual attack step (transition in the Petri net) is an object with its own properties as defined in the AKB. Each attack step will be applied on specific code fragments as described in the AKB, and as computed by the preceding attack steps in the Petri net.

In the next version of this deliverable (D4.03 in M24) once Task 4.2 and 4.3 will be already started, we will be able to estimate the time and probability of most of the attack steps, based on the following factors:

- Type of attack step (disassembling, static analysis, debugging,…)
- Code complexity and code metrics
- Attacker expertise (Amateur, Geek, Expert, Guru)

The AKB will then also be able to tell us for each attack step how much time will be needed by an attacker with a certain level of expertise to execute the individual attack steps, what their chances of success are, and what the outcome of their attack step is. For example, when an attacker prunes his search space as in step T9 in the above example, the AKB (which can use the information collected on the application's control flow graphs, slices, traces, etc) will allow us to estimate which code fragments the attacker will still be considering for his next attack steps.

On the basis of the individual times, probabilities, and outcomes obtained for the individual attack steps, we can then aggregate values of expected time and probability for the whole attacks by simulating the Petri net and performing the appropriate aggregation of values and properties.

In the Petri net of the OTP example, T4 and T5 are concurrent steps that will both need to be executed before T6 can be passed. So in the OTP example there are six paths leading to the final goal P10:

- Path 1: {T0, T1, T7, T8}
- Path 2: {T0, T2, T7, T8}
- Path 3: {T0, T1, T3, T4, T5, T6, T8}
- Path 4: {T0, T2, T3, T4, T5, T6, T8}
- Path 5: {T0, T1, T9, T10, T11}
- Path 6: {T0, T2, T9, T10, T11}

The aggregate time of a path will be the sum of the time to perform each attack step (transition) in that path on the code at hand.

$$\mathbf{Time}(\boldsymbol{expertise}, \boldsymbol{path_i}) = \sum_{\boldsymbol{all\ steps\ j\ on\ i}} \boldsymbol{time}(\boldsymbol{expertise}, \boldsymbol{attack\ step_j}, \boldsymbol{code_j})$$

The minimal attack time of the program is then obtained by computing the aggregate times over all parts, and taking the minimum of them.

The path with maximum probability instead can be calculated by iterating over all the possible attack paths in the Petri net, calculating the aggregate probability of each path: assuming that the attack steps are mutually independent (and therefore there is no correlation among them), the aggregate probability of a path will be the product of the probability of the attack steps (transitions) in the path: the path with the higher probability will represent the most likely path to succeed.

$$\mathbf{Prob}(\mathbf{expertise}, \boldsymbol{path_i}) = \prod_{\boldsymbol{all\ steps\ j\ on\ i}} \boldsymbol{prob}(\boldsymbol{expertise}, \boldsymbol{attack\ step_j}, \boldsymbol{code_j})$$

In the more general case, in which some of the attack steps are not independent, the joint probability will depend on the conditional probabilities between attack steps: thus joint probabilities can be computed after transforming the model into Bayesian networks [Jen96] or Markov chains [Dal06].

It is important to underline that in this general case, the probability of an attack step might depend on different attack steps previously performed but the temporal dependency expressed in the Petri-net does not necessarily imply a conditional probability between one attack step and its predecessor. Which forms of dependencies and probability models we need to support will become clear during the remainder of T4.1.

According to the security experts who wrote Section 5.3 in D1.02 on the ASPIRE Attack Model, Path 6 is the fastest for expert attackers as it is the less time-consuming for the attacker. Other types of attackers might lose time trying other paths or because they do not have the necessary expertise to implement the best attack path.

The time of each attack step will depend on different factors like attacker's expertise and code complexity. In the attack model, four levels of attacker's expertise have been identified (amateur, geek, expert, guru) and we can make the reasonable assumption that a higher-level attacker will take less time to perform one attack step than a lower-level attacker. Future work in tasks 4.3 and 4.4 of WP4 will be needed to better quantify this time with respect to each attacker's level.

About code complexity we can make the reasonable assumption that the more complex the code is, the more time an attacker will need to spend to perform an attack. Code size is the simplest form of code complexity, but perhaps not the most relevant one. Therefore alternative metrics for code complexity with regards to attack time will be evaluated in Task 4.2, where we will likely consider different types of metrics for different attack steps.

It is important to note that, as indicated in the above equations, not all attack steps take as input the same code. For example, the goal of step T10 in the OTP example is to identify the code executed before the OTP is shown, assuming that the search for consecutive XORs in step T11 can be shortened by focusing only on that part of the code. So clearly the effort and time needed for step T11 should be computed based on metrics applied to code selected in T10. It is for that reason that we will also need to model "the outcome" of each attack step.

2.5 Security Evaluation of Protected Applications

Once the attack paths are identified, we need to consider the effects of protections and measure their impact and effectiveness. We will do so by evaluating the additional time required to pass through Petri net models of the protected applications, compared to the times computed on the Petri nets modelling the vanilla application. We will, in other words, compute the delay that attackers incur as a result of applied protections.

Applying a particular configuration of protections with the ASPIRE tool-chain will change the Petri net model of an attack in several ways.

1. Protections will have (hopefully) increase the complexity of the code considered in each attack step, thus increasing the time to performing the step.
2. Likewise, protections can reduce the probability of success of the attack steps.
3. Protections might also change the relation between input and outcome of an attack step, for example by making it harder for an attacker to differentiate relevant code from irrelevant code for his attack step.
4. Protections can possibly also change the structure of the Petri net. For example when a protection is applied, an attack step might not be feasible anymore and the corresponding transition is removed from that new Petri net modelling the protected program version.

Using the modified nets, we can re-calculate the aggregate value of time and probability when one or more protections have been applied, and the difference of time and probability (before and after protections) will give an estimation of the effectiveness of a protection.

Considering the OTP attack example, we might want to add the following protections using the ASPIRE tool chain: (i) obfuscation of the code, (ii) anti-debugging code, and (iii) additional server-side checks to detect excessive numbers of provisioning requests.

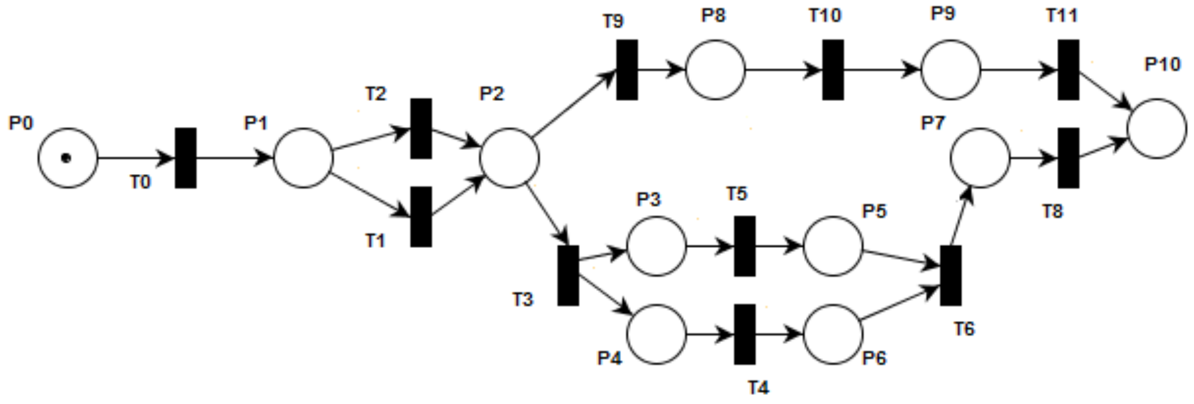

Figure 9: Protected version of the OTP Petri net

The new version of the Petri net in Figure 9 will automatically be extracted from the AKB when such a particular protection configuration is chosen. The new Petri net model will introduce a delay to attack steps T0 and T10, in which the attacker analyses the program to detect seed-processing code; anti-debugging can prevent dynamic analysis tools from working properly, thus introducing delays on attack steps T8, T9, and T11; server-side checks on provisioning requests make attack step T7 useless and hence that transition was deleted from the graph.

The effectiveness of a particular configuration of protections is then given by

Attack Delay =

MaxDelay (Protection configuration) – MaxDelay (Unprotected Application)

and by

Attack Success Ratio =

MaxProb(Protected configuration) / MaxProb(Unprotected Application)

In general, the higher the delay, the better is the chosen configuration of protections; and the smaller the attacker success ratio, the better is the chosen configuration of protections. In rare cases, higher delay might come witt higher success ratio. For that reason, the ADSS user will have to specify how to prioritize between the two features.

By simulating multiple Petri nets modelling various protection configurations, the ADSS will be able to select the most appropriate one. How the select the configurations will be researched in the remainder of the project.

2.6 Tools for Using Petri Nets

Petri nets are an important tool for the design and implementation of the security modelling and evaluation in ASPIRE. Petri nets have been extensively used in different domains and there exist a plethora of different tools to visualize and simulate Petri nets models, as listed in the on-line repository Petri Nets World [PNW].

In ASPIRE Petri nets will be mainly used tto insert attacks into the knowledge base and to simulate attacks automatically extracted from the knowledge base. So when selecting a tool, it is important to take into account the support for standard interchange format of the models, along with considerations about usability stability and user-base. We decided to narrow our search among the tools supporting the PNML (Petri Net Mark-up Language) standard format for representing Petri Nets models in an XML format defined by a precise standard meta-model [PNML] in the form of an EMF (Eclipse Modelling Framework) meta-model. In this way the mapping between the PNML's meta-model and the meta-model of the language used to represent the knowledge base, can be defined precisely and utilized to generate correct translators between the two languages.

Among the tools supporting PNML, we have currently chosen CPN-tools [CPN] as the main tool we will leverage to visually simulate Petri Nets models than can be imported/exported in PNML format. We will develop the translators from PNML to the ontology language. By using a standard format, we are not strongly bounded to a particular Petri Net tool.

Among other interesting PNML-based Petri-Net tools, we will consider ePNK [EPNK] and Coloane [COL], which are both designed as Eclipse plugin to be easy to extend and to integrate with other tools, and OWLS2PNML [OWL2PN], which provides a translator between PNML and OWL-S, an extension of OWL for service descriptions [OWLS]. More advanced time analysis and path analysis can be performed with TINA [TINA].

Regarding the evaluation of the probability in the simple case of mutually independent attack steps, CPN-tools might be configured and used to evaluate the attack path probability, but in case of conditional probabilities we might need to leverage on probability modelling tools, such as Netica [Net], AgenaRisk [Age], or OpenMarkov [OM].

2.7 Challenges and Open Research Problems

Obviously many challenges and research problems remain to be tackled regarding the foreseen use of Petri nets. These include:

- Developing the models in the AKB to allow the generation of Petri net models of all relevant attack paths. In literature, we can find examples of documented attacks and exploits, with (sometimes) detailed description and quantification of the amount of time and effort required to complete the overall attack (i.e., the followed attack path). Only rarely, however the detailed effort is reported for each attack task. Moreover, the values reported in literature refer to a few specific attack scenarios, they describe the effort required by a particular attacker, to port a specific attack on a given application. Even if such pieces of information are of fundamental importance to populate the AKB, because they come from real-world cases, they just represent data points that need to be generalized. This will be tackled further in this project in Task 4.1 of WP4.
- Developing the appropriate metrics that allow the Petri net simulation to compute relevant, representative attack times and success probabilities, both on protected and on unprotected applications. Therefore not only the dependence of attack time on code metrics needs to be modelled correctly, but also the influence that protections have on the metrics, given the metrics computed before protection, and given the limitations of the tool chain that will apply the protections. This will all be tackled further in this project in Task 4.2.
- Making sure that the modelling of time in terms of metrics is realistic. Code metrics are internal measurements, i.e. they just depend on qualities of the code such as code size. However, the security model needs to reason on external measurements, i.e., qualities of the code that involve humans, such as time-to-attack. Our assumptions on the relation among internal code metrics, attack tasks, attacker experience and protection configurations will be observed and verified in actual attack scenarios in Task T4.3 “Experiments with academic participants”, in task T4.4 “Experiments with industrial tiger teams” and in task T4.5 “Public challenge”.
- Hierarchical Petri nets induce a hierarchical relation between attacks. Thus, we should have a measurement of the attack’s “depth” or “abstraction level", which requires us to answer many questions. Do we need an attack abstraction classification, and if so, for which cases? What is our limit in attack depth (to which point of detail do we describe the attacks) What is our limit in abstraction level? Can we reach a depth where attacks can be re-used, as “lego blocks” to build more

complex attacks? What is the sweet spot between abstract classes of attacks, and very concrete attacks, e.g., relying on specific tools or even tool versions.

- The previous item relates to metrics: On the one hand, more concrete attack step modelling will allow more concrete metrics to be used and hence more accurate estimations of the required attack effort to be produced. But on the other hand, when the metrics become very specific for each concrete attack, splitting up a more abstract type of attack step into several more concrete steps introduces the risk that when an alternative concrete step is neglected (e.g., because some attack method is not known to the public or security researcher), a relevant metric will be overlooked. With more abstract attack step modelling, and consequently more abstract metrics, that changes become lower of overlooking relevant ones. This issue definitely will have to be taken into account when tuning our Petri net modelling.
- Linking attackers' level of expertise (i.e. amateur, geek, expert, guru as defined in D1.02) to all functions that map code metrics to attack time and success probability. A possible solution could be to simulate all nets for 4 separate scenarios (one per level of expertise) and mapping the colours red, orange and green used to summarize attack attributes in D1.02 to probabilities. For example, red could be associated to a high success probability (prob = 1), orange with an average probability (prob => 0.5), and green with a small probability (prob => 0.01).
- Ensuring that the necessary Petri nets are simulator to draw correct conclusions, yet avoid having to simulate too many. Given the number of protections, their configuration parameters, the possible selections of code fragments to apply them on, etc., the number of possible Petri nets to be simulated can become intractable. Solutions we will investigate include sampling, hierarchical modelling, and parameterizable simulation. The latter might be useful because we foresee that many protections will change the parameters and inputs of functions that map metrics to time and probability, without changing the actual structure of the Petri nets themselves. For example, when a protection makes an attack step useless or impossible, rather than removing the paths involving that step from the Petri net, we could also model its attack time as infinite, or its probability as zero. What the best option and combination of options will be, will be researched mainly in Task 5.2 of WP5, when an initial but functional ADSS becomes available for experimentation.
- Ensuring that the ADSS can still produce useful quantitative results, albeit perhaps incomplete, in case the timing or probability function of some attack steps cannot be estimated or modelled reliably. We foresee several ways in which this will be possible. For example, by assigning a worst-case attack time of zero to such attack steps, the Petri net simulation will already be able to decide whether the steps are on the fastest attack path or not. If they are not, not knowing their exact behaviour is no issue. If they are on the fastest path, the ADSS can then try to push them off the fastest path by deploying protections that specifically impact the preceding or following attack steps on that fastest path, again making the lack of an exact model a non-issue. Finally, even in case the ADSS will not be able of estimating attack times reliably, it may still be able to provide a rough estimation of the attack delay incurred by some protections: if the MaxDelay computed on the vanilla application (which is basically a sum of attack step times) includes some unknown or only roughly known x, and the MaxDelay computed on the protected application includes x + 50 instead, it is still possible to compute the attack delay of x + 50 - x = 50. Options like these will be considered mainly in Task 5.2 of WP5, but they will of course also affect the focus of experiments in T4.2.

Section 3 The ASPIRE Knowledge Base

Chapter Authors:

Cataldo Basile (POLITO), Bjorn De Sutter (UGent), Paolo Falcarin (UEL)

3.1 Knowledge to be Modelled

From the introduction in Section 1, we briefly recall that the ADSS will rely on the knowledge in the AKB to identify the attacks that attackers can mount against applications and their assets, and to identify the protections that can be enforced to mitigate the threats of those attacks. Using this knowledge, and taking as inputs an application and description of its assets, the ADSS will select and generate the appropriate Petri nets to simulate in order to evaluate and measure the level of protection that can be provided by the available protections to mitigate the threats.

The ultimate goal is to enable the identification and selection of the best measurable protections.

Information in the AKB can be classified in two categories: generic knowledge and execution-specific knowledge.

A priori knowledge, indicated as Generic Knowledge in Figure 2, comprises information that is independent from the program to protect. It includes (but it is not limited to):

- 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, types of security properties to protect in applications;

The ASPIRE knowledge base needs to describe the **concepts** (e.g., attacks, protections), their **relations** (e.g., attacks can undo or work around protections, protections can mitigate attacks), and a **characterization** of those concepts (as informally documented in literature and in ASPIRE D1.02). Roughly speaking, the AKB will contain classes to precisely describe the concepts listed before, a complete object instantiation of those classes (that will be initially done based on D1.02 and then maintained by Consortium partners), and all the associations needed to relate instances of the concepts.

Additionally, a priori knowledge has to contain enough information to provide a means to instantiate the following data:

- metrics to evaluate the probability of successful attacks, delays etc.;
- several abstract representations of the target application on which the metrics can be computed, including source code, object code, control flow graphs, program dependency graphs, data dependency graphs, call graphs, etc.;

Additionally, a priori knowledge includes execution platform and other software-related data that are independent of the fact that our aim is protecting software. Therefore, it will include concepts like OSes, processors and their instructions sets, emulators, interpreters, which can be used to decide on the feasibility of some attacks and applicability of some techniques. Additionally, a priori knowledge includes compilers, linkers and their options, as they may affect the performance and structure of the code and may also be incompatible with some protection tools.

It is worth mentioning that all concepts are independent of the ASPIRE project, and hence generic, even if their modelling will be focused on the ASPIRE decision support.

Additionally, a priori knowledge includes many other concepts that only exist in the ASPIRE scope, for instance:

- ADSS-related information, like ADSS-specific preferences, optimization strategies (i.e., what objective to optimize to select the best protection), pruning strategies (i.e., how to reduce the size of the problem space to be computationally feasible), etc.;
- ASPIRE tool chain and tool chain components, that is, the abstract description of the tools that will be actually used to implement the protection which must allow ADSS to select the tools that will actually implement the protections, to decide which options of the tools to enable, and to choose the proper value for the tool options.
- ASPIRE annotations, added by the software developer to mark assets and other critical sections of the application, and other formats used by the ASPIRE tool chain components to exchange data or to annotate code.

Execution-specific knowledge contains all the user[2] and application centric information needed to express the target application and assets the user wants to protect, and user preferences on how to protect that target application. Execution-specific knowledge complements the generic a priori knowledge to permit the ADSS to

- decide which is the best protection to apply on the target application,
- to precisely understand what to protect,
- as well as how to drive the tool chain components when actually implementing the protections.

Roughly speaking, execution-specific knowledge is the instantiation of the abstract models that are part of the a priori knowledge, and the user-provided information needed to generate that instantiation. Therefore, execution-specific knowledge includes (but it is not limited to):

- the target application and its application "components", (like functions, procedures, stubs)
- the assets and the corresponding threats as provided by the user;
- abstract representations of the application and its components, such as control flow graphs, data dependency graphs, etc. on which metrics can be computed; these representations will cover the original application as well as the protected versions being simulated, and the derivations thereof in different phases of simulated attacks (such as when an attack step is considered to have been applied that undoes part of the applied obfuscations);
- the execution environment where the application will run, the used compiler and linker. This information is repeated for all the executing components (like client and server stubs, or the nodes for a distributed computing);
- user ADSS characterization, that is the ASPIRE tool chain components available at the user installation[3];

[2] In this section, as in the remainder of this document, the terms user and developer both refer to people responsible for providing the software. The terms are used interchangeably, albeit that "developer" typically refers to the author of the software (and of the annotations in it) to be protected, whereas "user" refers to the user of the ADSS and the ASPIRE tool flow, i.e., the person responsible for applying the protections onto the developer's software.

[3] As we foresee different ADSS profiles for exploitation purposes, one user may have the open source version, where all the tool chain components are open source, another one may have the premium version, that also includes proprietary components, and others may have just bought individual tool

- the state of the current tool chain execution and all the output from already executed components;
- attacks and protection relations with the program asset and user preferences.

The AKB will keep track of the state of the tool chain to let the ADSS check that all foreseen protections have indeed been applied, and possibly also to tune the configuration of protections applied later in the tool chain to take into account the results of previously applied ones; the output will be used to generate a report for the user that links the ADSS decision making with the logs produced by the compiler (as needed according to requirement REQ-ASR-005 of ASPIRE deliverable D1.03).

The dynamic nature of this tool chain execution state and of the representations of program versions being simulated already hints to the fact that the AKB will not be a static knowledge base that simply combines the a priori knowledge and the execution-specific knowledge. Instead, it will be a dynamic knowledge base that is continuously enriched as more knowledge becomes available during the operation of the ADSS.

3.2 Roadmap for the Knowledge Base Definition

The design of the ASPIRE Security Model and the ASPIRE Knowledge Base is conducted in phases:

1. Informal definition of the threat model (M3, already addressed in D1.02).
2. Initial definition of the conceptual model of the ASPIRE knowledge base (M6, one of the main focuses of this document).
3. Definition of the attack simulation models and support from the ASPIRE Knowledge base (M6, one the main focuses of this document).
4. Formal definition of knowledge base enrichment (M9, D5.01).
5. Intermediate update of conceptual model and the attack simulation models for a more in-depth definition of metrics (M12, D4.02).
6. Final delivery of the security model (M24 with D4.03 for the conceptual definition, D5.07 for the knowledge base enrichment).

The conceptual model of the ASPIRE knowledge base comprises all the formal models that permit to statically describe the information required by the ADSS to work. Since we are at the beginning of the project, the ADSS architecture is only sketched in internal documents and likely subject to changes. Additionally, there are many other indispensable things for the conceptual model that will only be produced during next months. For instance, we mention here the ASPIRE tool chain, the characteristics of applications to protect that need to be considered when evaluating the actual protection level and the overhead given by the application, the impact of software protections on the asset in the target application, exact relations and effects of the consecutive use of two or more protections on the same application, and so on. Moreover, some of the most innovative parts of the project – that we really want to be supported by the ADSS – will be delivered only later.

Therefore, the exact definition of “the information required by the ADSS to work” will be known only later. However, starting from D1.02 and from our experience in software protection, we can already sketch an initial conceptual model and identify the macro areas that need to be covered in future months. The purpose of this section is to present the initial

chain components. Moreover, also during the project itself, the tool chain will evolve as foreseen in the project DoW, with gradually more protections being supported. Rather than updating the decision logic in the ADSS to reflect changes in the tool flow, we make the description of the available protections part of the AKB.

conceptual model of the ASPIRE knowledge base in a format that is independent of any technology used to actually represent it (phase 2). Needless to say, the initial conceptual model has been designed to be expansible, that is, there is no need to rework the entire model to apply any change.

For this reason, the conceptual model has been structured to be composed of a main model and a set of sub-models. The main model introduces the principal concepts from D1.02 and formally relates them. For this reason, we consider the main model relatively stable. Sub-models detail each of the concepts in the main model, thus presenting a clear snapshot of the knowledge related to the concept they expand. The sub-models presented in the intial conceptual model cannot yet be considered stable, for the reasons presented above. Section 3.3 will present the main model; Section 3.4 will present the preliminary description of the principal sub-models. This means that a significant part of the a priori knowledge in the ASPIRE knowledge base is described in the initial conceptual model. Actual knowledge about concrete attack paths, protections, relations between them, etc. will be described manually by ASPIRE partners and added to the knowledge base as it becomes available as output of WP4 task and result of the ADSS design (phases 4-6).

Exactly how much and how concrete this manually added knowledge will need to be is an open research question at this point in time. In practice, this information will be based on existing attack examples, such as the attack paths described in Section 5 of D1.02, successful attacks published in literature in the past or being known to the partners, e.g., with regard to their own products. The sources of the information will hence be fragmented. However, when all this fragmented information is combined, it needs (1) to be relevant and applicable to a broader range of scenarios than the examples at the information's origin, and it needs (2) to be free of inconsistencies and duplicated work.

For that reason, we aim at reducing the manual effort by (1) designing the model in such a way that all information that will be modelled manually, will be modelled at the right level of abstraction and granularity (which will be determined in the coming months), and (2) by using reasoning engines to complete the thus modelled a priori knowledge with all information that can be deducted from it. Therefore an enrichment process will be added on top of the conceptual model. The result of the manual preparation and the enrichment on the manually provided data will then form the a priori knowledge that will be shared among all the protections.

As introduced in Section 1.2, execution-specific knowledge is initially instantiated from the information obtained from analysing the target application. This information must be automatically added to the knowledge base by importing it. For example, the control flow graphs can be produced by static analysis tools and imported. This activity is depicted in Figure 2 as the ASPIRE Program Analyses.

Also regarding execution-specific knowledge, enrichment will help in completing the information. For execution-specific knowledge enrichments in fact plays a major role. Starting from general information on attacks, protections, and relations to assets, it will infer protections and combination of protections that will work on the target application, given the assets and security properties requested by the user (the Scenario in Figure 2).

Also in this regard, the precise definition of which information will be needed by the ADSS is not fully specified at the time of writing this report. We foresee to deliver the definition of the enrichment framework both for the a priori knowledge and for the execution-specific knowledge in phase 4. Hence, a main design requirement for the preliminary enrichment framework as presented in this document is (again) expansibility.

The ASPIRE knowledge base contains static information. It is not practical, however, to to use static information to evaluate the feasibility of some attacks against the target applications. The reasons are the extreme variability of the applications to protect, the impacts protections may have on applications, etc. This is why we introduced the attack simulation models (phase 3). From the knowledge base point of view, the use of attack simulation models creates two new requirements: first, the models to simulate must be derived from the information stored in the ASPIRE knowledge base including the simulation of vanilla and protected applications. Secondly, the output of the simulation must be stored in the ASPIRE knowledge base to be used by the Selection & Optimization process, as already described in Section 1.2.

We will therefore elaborate an important sub-model that describes metrics and how they can be computed on vanilla and protected applications. In this document, a very preliminary definition of metrics is being presented. Since tasks related to metrics (mainly T4.2, but also T4.3, T4.4) did not start yet, results in that sub-model cannot be considered mature. For this reason, metrics are not part of the initial version of the main model. Nevertheless, this important concept is eligible to be part of the main model for future versions of the conceptual model. Deliverable D4.02 (M12) will serve to fill this gap in the initial version of the conceptual model: D4.02 will report an updated version of the conceptual model that details on metrics (phase 5).

The final version of the ASPIRE knowledge base will be documented in two deliverables: D4.03 (M18) will present the final conceptual model; D5.07 (M24) will present the final enrichment framework. By taking advantage of the expansibility of both the conceptual model and of the enrichment framework, possible changes that may be required during the last year will be easily supported.

3.3 The Main Model

This section presents a class diagram that represents a formalization of the attack model information represented in D1.02. That attack model included the following high-level concepts and informal relations between them:

- Application;
- Asset;
- Attack;
- Attack path;
- Attacker;
- Software protection;
- Tools.

Figure 10 presents the UML class diagram that shows how these concepts have been formally related. It is worth noting that associations are all many-to-many associations unless differently noted in the text.

First of all, we present the `Application` class. Its instances will be objects abstracting the applications to protect. Next, the class `Asset` describes the assets. A preliminary set of assets has been listed in Section 3 of D1.02. The identified assets in our formal model will be detailed in Section 3.4.2 Presently, the set of assets we identified suffices to describe the types of assets listed in D1.02. Nevertheless, we do not exclude the possibility of updates. Application instances are associated to at least one `Asset` instance by means of the `contains` association.

In most cases, we must consider not the whole application but one of its parts. For this purpose, we introduced the `ApplicationPart` class, whose instances may describe logically self-contained components (like server and client stubs, algorithms) or simple pieces of code or similar abstractions (like fragments and slices). `ApplicationPart` instances are connected to the Application instances they are part of by means of the one-to-many hasPart association. Additionally, ApplicationPart instances are associated to Asset instances by means of the partContains association. More details on ApplicationPart class, its subclasses and associations will be presented in Section 3.4.1 where the Application sub-model is presented.

Assets are associated to several security properties a user may be interested in when protecting the asset. This information is conveyed via the AssetProperty class instances (named threats in D1.02) that are associated to Asset instances by means of the one-to-many hasProperty association. Additionally, assets may depend on other assets. The (self) association requires is used to represent this dependency. We anticipate that the information needed to construct the associations between applications, their assets and the threats to protect against, will be obtained from the developer's source code annotations and through additional information passed to the ADSS.

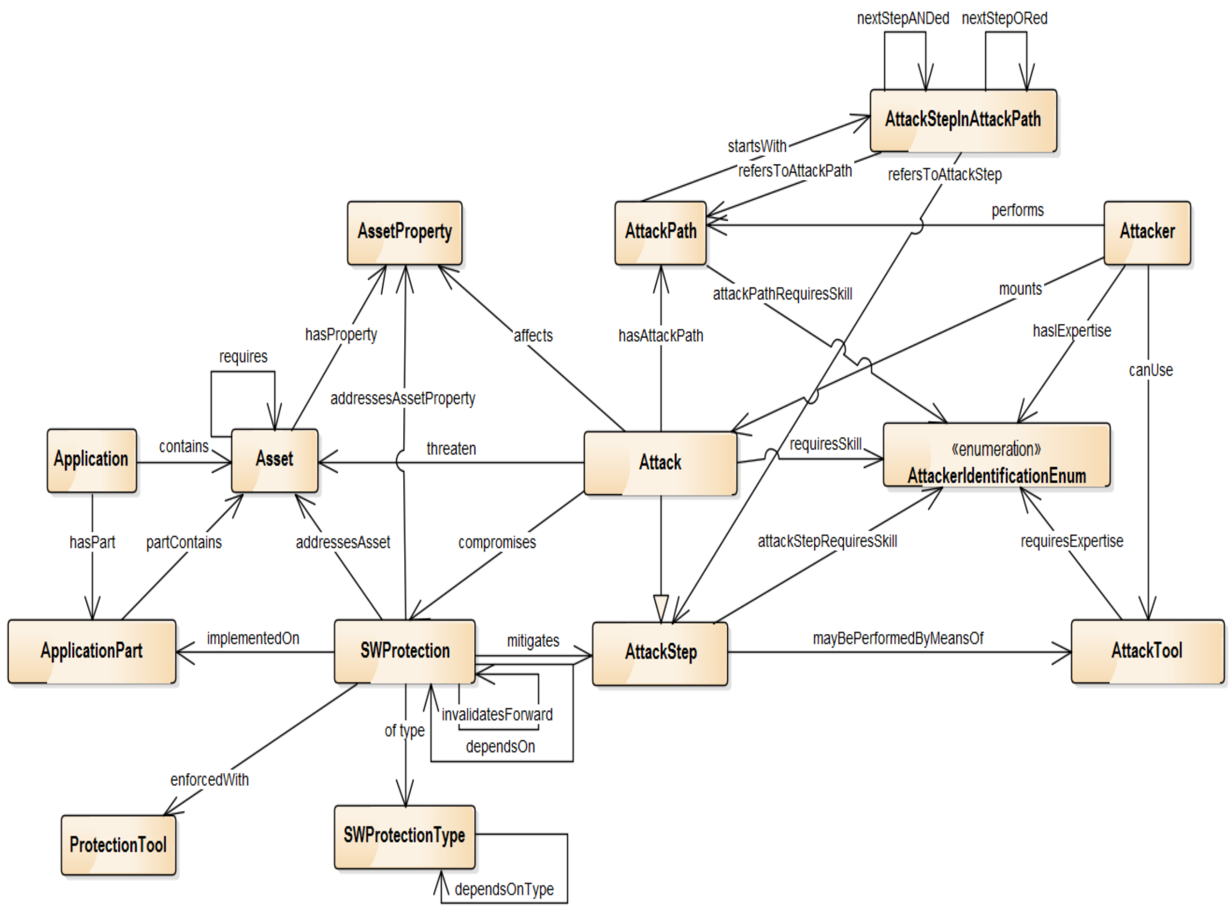

Figure 10: Main model

Attacks are represented as instances of the Attack class. Several attacks may threaten an asset, this scenario is represented by means of the threatens association. Additionally, attacks may compromise asset security properties, this is represented by means of the affects association.

Having modelled the attacks, we are able to model attack paths, and naturally, we use the AttackPath class for this purpose. To represent that an attack can be mounted following an attack path we use the hasAttackPath association that relates Attack instances to AttackPath instances (one-to-many). Attack paths are decomposed in individual attack steps that are represented as AttackStep instances. In theory, each attack step can be an entire attack on its own. We depicted this scenario by means of the inheritance, that is, Attack is a subclass of AttackStep (bounded via an 'is a' association). This also means that we can describe steps that are not attacks per se (as do not affect any asset property) as they are only needed within an attack path. Each AttackStep instance is associated to the tools that can be used to actually mount it. Attack tools are modelled with the AttackTool class and AttackTool instances are associated to AttackStep instances via the mayBePerformedByMeansOf association.

Since attack steps can be shared among different attack paths, we used the AttackStepInAttackPath (association) class, which links the AttackStep instance via the refersToAttackPath association and the AttackPath via the refersToAttackStep association. The last two associations bind an AttackStepInAttackPath instance to exactly one AttackStep instance and one AttackPath instance. Then, to describe consecutive attack steps we used the nextStepANDed and nextStepORed associations that serve to also indicate if the consecutive steps must be all executed or at least one must be executed. The first step of the attack path is indicated by setting up the startsWith association. It is worth noting that nextStepANDed, nextStepORed, and startsWith are many-to-many associations as many parallel steps can be performed at the same time (see Section 2), that is, these associations allow us to model graphs of attack steps (and Petri nets), not only sequences (see Section 3.4.4).

Attacks are categorized by the level of exploitation (see Section 4.2 of D1.02). As presented in D1.02, attacks may require different expertise, skill and resources to be mounted, and attacks may have different levels of exploitation. D1.02 introduced the “identification” concept (and distinguished four categories of attackers: gurus, experts, geeks, and amateurs), and the “exploitation” attribute (and distinguished three categories of exploitation: low, medium, high). We model these formally with two enumerations: the AttackerIdentificationEnum class with the four identified attacker categories, and the ExploitationEnum class with the three identified exploitation levels. We don’t expect changes to these enumerations, however it is worth noting how easy it is to extend or modify these classifications. AttackStep instances (and Attack instances as well due inheritance) are associated to AttackerIdentificationEnum values by means of the requiresSkill association and to ExploitationEnum values by means of hasExploitation association. An attack step is associated to exactly one AttackerIdentificationEnum value and exactly one ExploitationEnum value. Also AttackPath instances and AttackTool instances are associated with AttackerIdentificationEnum instances via the attackPathRequiresSkill and requiresExpertise associations.

Software protections are described by means of instances of the SWProtection class. To refer to the tools actually deploying protection (which we aim at configuring as output of ASPIRE DSS), SWProtection instances are associated to ProtectionTool class instances by means of the enforcedWith association. To indicate possible dependencies among software protections, the dependsOn association is used, while forward incompatibility is represented through the invalidatesForward association. That will allow us to model when one protection cannot be enforced after another one because it renders the previous one useless or wrong, like obfuscating code after having inserted guards, or when the first protection invalidates the pre-conditions to apply the later one.

Software protections are categorized in types by associating them to SWProtectionType instances via the ofType association. A software protection should be associated to exactly one SWProtectionType instance. We will evaluate if some protection could need more than one association in the next months (e.g., remote attestation implemented via renewability like in [Fal06]). Together with the need of categorizing protections, this class serves to map the ‘lines of defence’ concept, as presented in the DoW (see also Section 3.4.3). Another reason for having this class is that it will allow class-level reasoning about protections, i.e., to answer questions like “which are the categories of protections that can be used in combination with strengthen local integrity protections to increase the protection level?”

Dependencies among `SWProtectionType` instances are represented by means of the `dependsOnType` association.

`SWProtection` instances are related to the `Asset` instances (via the `addressesAsset` association) and `AssetProperty` instances (via the `addressesAssetProperty` association) that they may mitigate risks on, and to the `Attack` instances that they aim at invalidating or making more difficult (via the `mitigates` association). By means of the `compromises` association, attacks are associated to the `SWProtection` instances they can invalidate (or render useless or completely remove).

Attacks are mounted by attackers, represented by instances of the `Attacker` class, which are related to the attacks they may be interested in performing by means of the `mounts` association. We just sketch in the main model the relations of the Attacker class, which will be presented more in details in Section 3.4.6 where the protection requirements sub-model will be presented. To relate attackers to the attack they can mount, `Attacker` instances are associated via the `hasExpertise` association to the `AttackerIdentificationEnum` values. The attack paths attackers can mount are represented via the `performs` association. Attacks are strictly related to the tools attackers use to actually perform an attack step. `Attacker` instances are thus related to `AttackTool` instances using the `canUse` association. As anticipated before, to be very precise, in the security model, each `AttackStep` instance has been associated to `AttackTools` instances that can be used to mount the attack by means of the `mayBePerformedByMeansOf` association. At present, we don't expect to model single attackers, we just need the flexibility to express that various attacks can be mounted by several attackers that may have different expertise and different tastes on the attack paths to follow to mount an attack (see Section 3.4.6).

3.4 Model Extensions: the Sub-Models

The main model presented above depicts the main concepts and their (high-level) relations. However, a refinement is needed to allow a more fine-grained description of the protection scenarios ASPIRE has to face. The main tool we use to refine the main model is inheritance. It helps us to refine main model concepts and at the same time preserve the associations among them.

Refinements to the main model will be presented by means of a set of sub-models that cover six areas:

- **Assets**, which describe what to protect and report categorization already introduced in D1.02.
- **Applications and their execution environments**, which precisely characterize the application to protect. This will allow us to better target the protection protection and to provide information about the execution environment that may determine classes of attacks and protections. For example, attacks against an applications running on Windows may be different from the ones the same application has to face on MacOS, and some protection techniques cannot be available on all the platforms.
- **Protections and protection types**, which define a taxonomy of the different protections.
- **Attacks**, which describe the attacks that can be mounted against applications, including the attacks already identified in D1.02.
- **Metrics**, which provide a very preliminary identification of the classes to use to convey information about metrics and the general types of metrics-related classes.

- **Protection requirements**, which capture our initial ideas on how to specify protection requirements on the target application, and which complete the information presented in D1.03.

The areas and the corresponding list of sub-models is not definitive, as these are the areas we currently identified. The next sections will present the initial definition of these sub-models.

3.4.1 Application sub-model

The sub-model shown in Figure 11 presents an initial definition of information about the target application that the ADSS will require to evaluate the impact of protections and thus decide on the best protection according to user requirements.

We highlight five major concepts, which will be developed into separate packages:

- The **parts** and **components** the application is made of, described by means of the `ApplicationPart` class and detailed in the `ApplicationParts` package.
- The **platform** where the target application or some of its components will be executed, such as client and server stubs. This is described by means of the `ExecutionEnvironment` class and detailed in the `ExecutionEnvironment` package;
- The possible **communications** between two or more of the application components described by means of the `Communication` class and detailed in the `Communications` package.
- The way the source code is **compiled and linked** to obtain the executables, described by means of the `Compiler` and `Linker` classes, and detailed in the `Compilers&Linkers` package.
- The types of **abstract representations** of the target application that could be needed to evaluate the impact of protections, select the best protections, and actually enforce them, described by means of the `Representation` class and detailed in the `Representations` package.

These five packages will be independently developed during the next months. This list of packages is also preliminary. Other packages could be added depending on the ADSS design.

From Figure 11 it is possible to see that an `Application` instance is connected to:

- Its application components and parts, instances of the `ApplicationPart` class, via the `hasPart` association, that has been already illustrated in the main model.
- The `ExecutionEnvironment` instances in which the application can be run via the `executedIn` association.
- The `Representation` instances, that abstractly describe these application parts via the one-to-many `hasRepresentation` association;
- The `Compiler` and `Linker` instances use to compile it, via the `compiledWith` and `linkedWith` assocations, which are valid if the single application components are not compiled independently.
- The `Communication` instances of which the `Application` is an endpoint, will be identified by navigating the `endpoint1` and `endpoint2` associations in opposite direction.

It is worth noting that `ApplicationPart` instances are also linked with the same concepts are `Application` instances in the `ApplicationParts` package. The `ApplicationParts` package presented in Figure 12 describes the parts of an application that are of interest for the protection, for instance because they contain assets, are targeted by a protection, or are the endpoints of some secure communication channel. We identified several concepts and described them by means of inheritance. We consider the following subclasses:

- `Function`, and `Procedure` classes, whose instances explicitly declared functions, procedures, and methods in the target application.
- `Algorithm` class, whose instances are algorithms, which may be formed of several `Function` and `Procedure` instances (associated by means of the `includesAlgorithm` and `includesProcedure` assocations).
- `Slice` and `Fragment` classes, whose instances represent parts of the code not necessarily corresponding to entire functions and procedures. Examples are the barrier slices (investigated in WP2) and the server and client parts generated during code splitting, represented by `ServerStub` and `ClientStub` instances. `Fragment` instances will describe sets of consecutive statements and will be needed to describe metrics (see Section 3.4.5).
- `Library` class, used to describe static or dynamic libraries, represented as `StaticLibrary` and `DynamicLibrary` class instances. `DynamicLibrary` instances are connected to `LibraryModule` instances by means of the `containsLibraryModule` association.
- `StaticallyAllocatedData`, used to describe (security-sensitive) data embedded in the executable files or libraries, such as cryptographic keys (represented by means of the CryptoKey class), initialization values for tables (represented by means of the TablesInitiValues class), etc.. A notable example of statically allocated data are strings, which attackers look after during certain attack steps, modelled with an ad hoc subclass `String`.
- The `File` class is used to model external files and data, such as custom configuration files (represented by means of the ConfigurationFile class), or registries/manifest files that might contain relevant data to be taken into account (represented by means of the Manifest class).
- `Class` is the only element presented in this initial characterization of application components originating from object-oriented programming. We added this mainly as a placeholder to remember us that object-oriented programming constructs and entities need to be considered while extending or adapting the models. Because of resource limitations, however, ASPIRE will only implement techniques for C code for the time being.

The `ApplicationPart` instances are connected to:

- the `Application` instance they are part of (via the one to main `hasPart` association),
- the `ExecutionEnvironment` instances where they can be run via the `componentExecutedWIth` association (as some application is made of several independently executing component, like clients and servers),
- the `Communication` instances they are endpoint of (via the one-to-many `endpoint1` and `endpoint2` associations)
- the `Representation` instances, that abstractly describe these application parts (via the one-to-many `componentHasRepresentation` association;
- if they are independently compiled and linked, the `Compiler` and `Linker` instances via the `componentCompiledWith` and `componentLinkedWith` associations.

Note that we preferred the use of the work “component” instead of “part” for the associations that are logically self-consistent like executable portions of the application (like clients and servers) or independently compiled portions of the application.

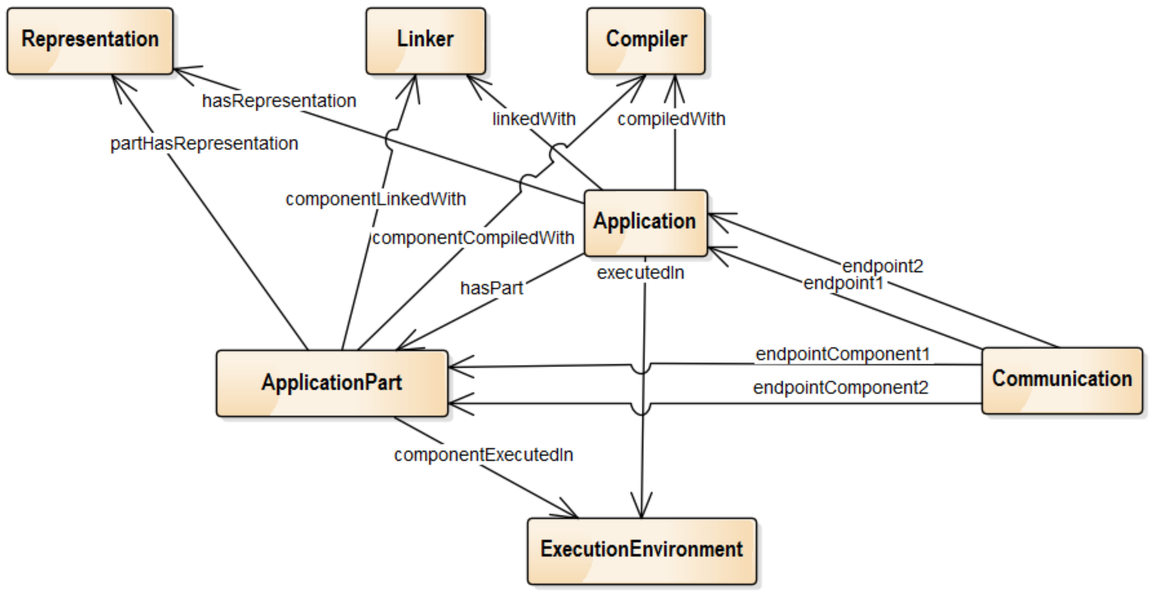

Figure 11: The Application sub-model.

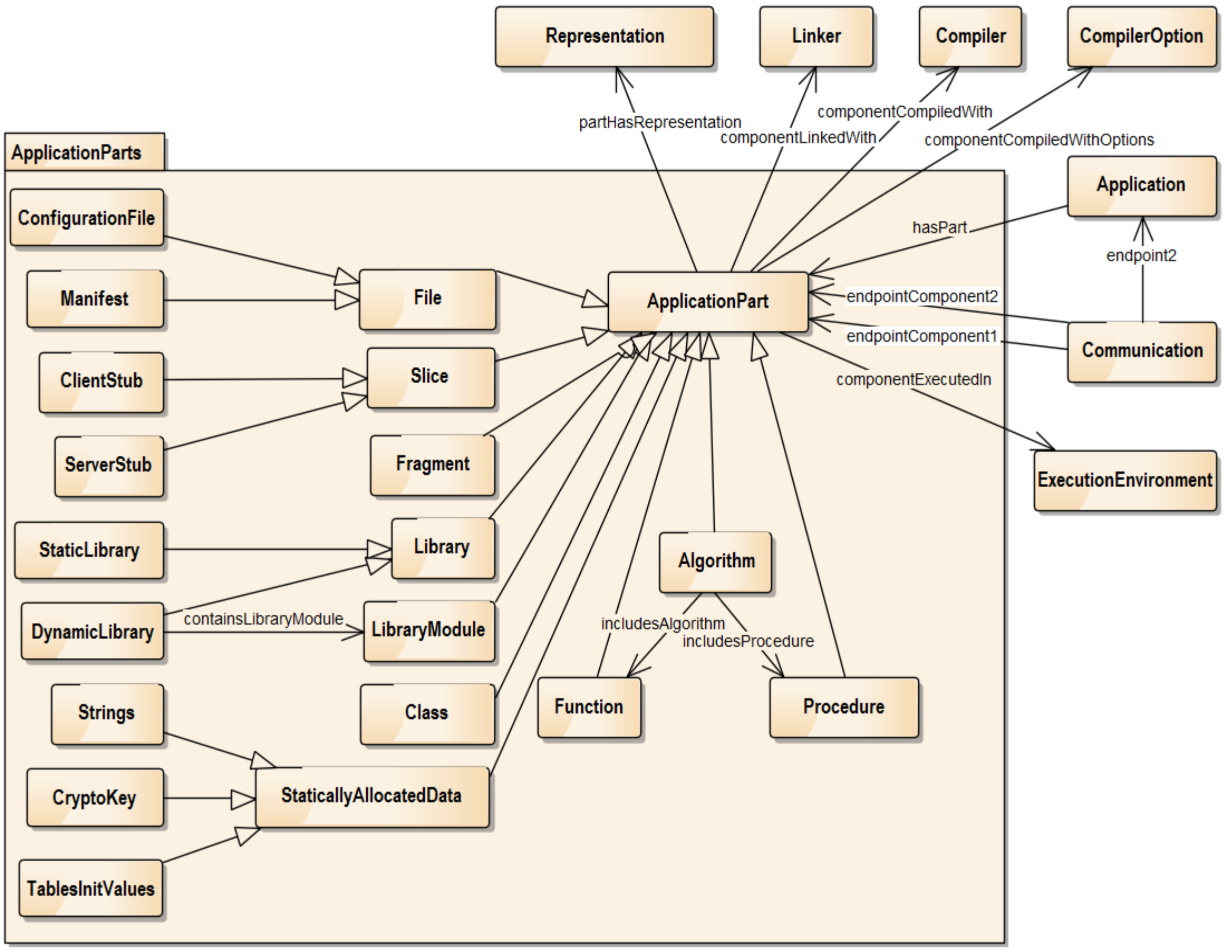

Figure 12: The `ApplicationParts` package.

The `Representations` package (see Figure 13) will include all the abstract representations that can be used by the ADSS or tool chain components. It includes the abstract `Representation` class which will be subclassed any time a new abstract representation will be of interest of the ASPIRE project. Currently, we included the following subclasses:

- `SourceCode`, and `ObjectCode`, whose instances are self-explaining.
- `ControlFlowGraph`, a graph representation that models how control can be transferred in the procedure or program, i.e., in which orders instructions can be executed.
- `ControlDependencyGraph`, a graph representation that models to what extent the execution of instructions in a procedure or program depends on the execution of the other instructions in that procedure or program.
- `DataFlowGraph`, a graph that models how values computed in the program are computed out of other values computed (elsewhere).
- `ProgramDependenceGraph`, a graph that combines the data flow graph and control dependency graph presentations to model how the execution of instructions depends on other instructions being executed and on the values being computed by those other instructions.
- CallGraphOfProcedure, a graph representation that models which procedures in a program can call with procedures.
- `Annotation`, that will be used to report the annotations added by the developers or by tool chain components to tag pieces of code.

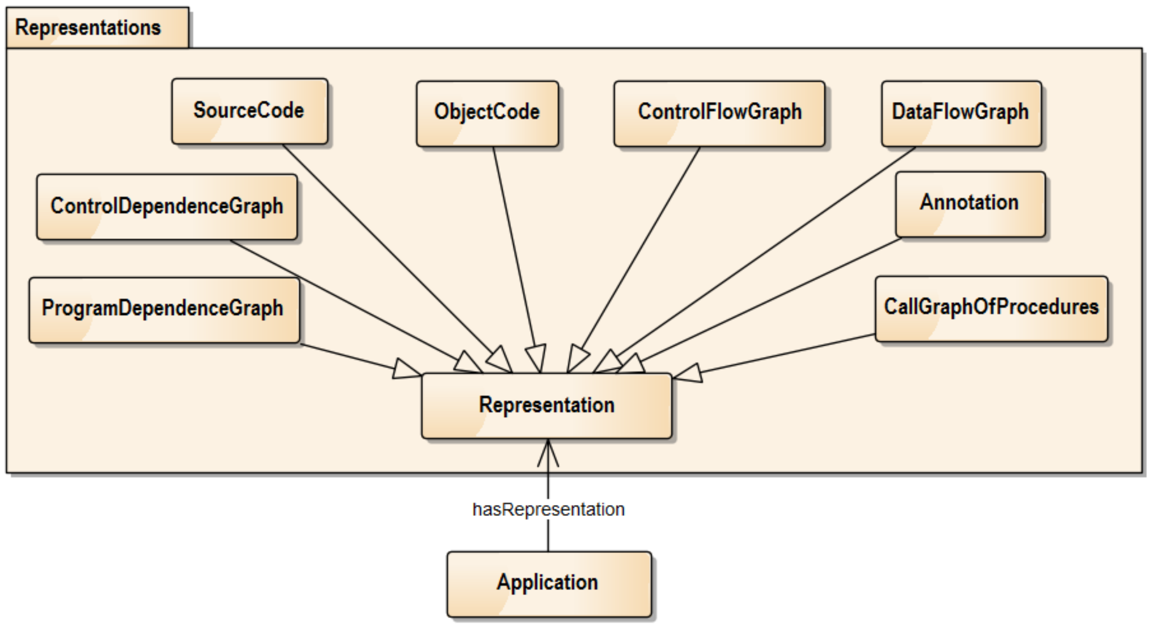

Figure 13: The `Representations` package.

`Representation` instances are associated not only to `Application` instances, via the one-to-many `hasRepresentation` association, but also to `ApplicationPart` instances, via the one-to-many `componentHasRepresentation` association (as abstract representations are in most cases created for components, like functions and procedures).

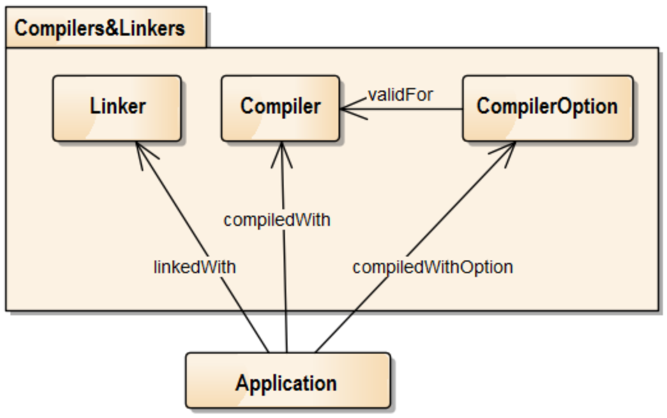

Figure 14: The `Compilers&Linkers` package.

For the `Compilers&Linkers` package depicted in Figure 14, we highlighted two main concepts: compilers, described as instances of the `Compiler` class, and linkers, described as instances of the `Linker` class. `Application` instances are associated to `Compiler` and `Linker` class instances via the `compiledWith` and `linkedWith` one-to-many associations, and to `ApplicationPart` instances via the `compiledWith` and `linkedWith` one-to-many associations.

For the ASPIRE project, it is important to model the options/flags that can be selected during compilation because they can affect the performance and structure of the code and may also be incompatible with the processes performed by some protection tools. For example, Diablo requires the availability of separate object files and a map file of the linked binary or library, all of which can be generated by employing the appropriate options on existing compilers. Furthermore, to select the most appropriate protections to be applied, it can be useful to know, e.g., whether an asset in the form of a code fragment is duplicated because of compiler optimizations such as inlining. For these reasons, we have foreseen the `CompilerOption` class.

Application and `ApplicationPart` instances are bound to the options used to compile them by means of the `compiledWithOption` and `componentCompiledWithOption` associations. We want to note that `CompilerOption` instances are independent of the compilers. Each of them models (equivalent) options that can be invoked with different commands on different compilers. So each option is represented as a single `CompilerOption` instance and associated via the `validFor` association to the `Compiler` instances where they are available. The reason for this design approach is that the alternative method, i.e., creating instances of all the options for each compiler and then explicitly requiring their equivalence, is less efficient from the reasoning point of view.

Other code manipulations could also be of interest in this package. In particular, the description of the optimizations that are performed by the compilers can be interesting to derive some general information of the object produced and may serve the decision process. The ASPIRE Consortium is still debating the need to include this information in this package.

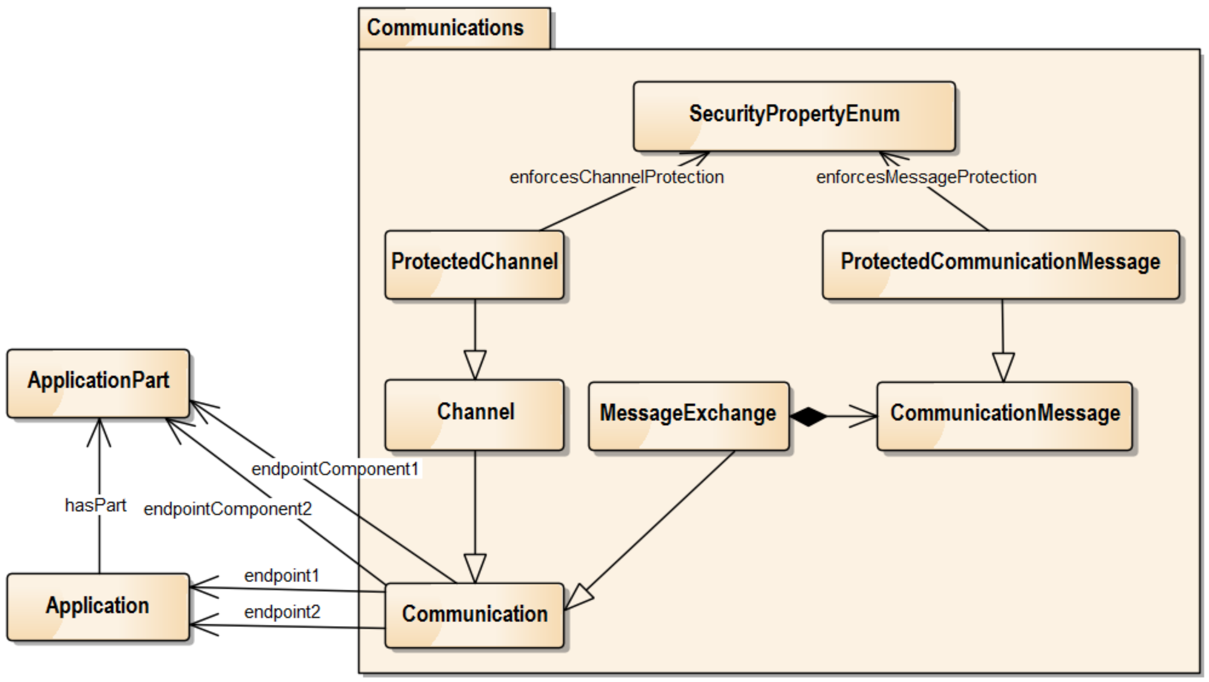

Figure 15: The `Communications` package.

Figure 15 displays the structure of the `Communications` package. Its main class is the abstract `Communication` class. We foresee two types of communications, channel-oriented and message-oriented communications. Therefore `Communication` has been subclassed in `Channel` and `MessageExchange` classes. The `Channel` class has been further subclassed in the `ProtectedChannel` class to describe protected channels. Instances of the `ProtectedChannel` class are bounded to the security properties they guarantee by means of the `enforcesChannelProtection` association. Security properties are represented by `SecurityPropertyEnum` instances, an enumeration class listing all the possible security properties that can be enforced on communications (e.g., integrity, confidentiality, replay protection). A `MessageExchange` instance is composed of a set of messages, represented as `CommunicationMessage` instances. To mark protected messages, i.e., messages that have been processed to ensure some security property (like message authentication, integrity, and confidentiality), we use `ProtectedCommunicationMessage. Its` instances are also connected to `SecurityPropertyEnum` instances, via the `enforcesMessageProtection` association.

Communications are characterized by their endpoints. In order to support non-oriented communications as well, endpoints are described with the one-to-many `endpoint1` and `endpoint2` associations from `Communication` instances to `Application` instances. In case we want to describe oriented communication, the application referenced by the `endpoint1` association can be considered the originator and the application referenced by the `endpoint2` association the consumer. It is worth adding that this enables the description of communications between `ApplicationPart` instances, e.g., when their components are to be executed on different machines, such as the different forms of stubs that will execute on the client and on the server. For that reason, we introduced the one-to-many `endpointComponent1` and `endpointComponent2` associations.

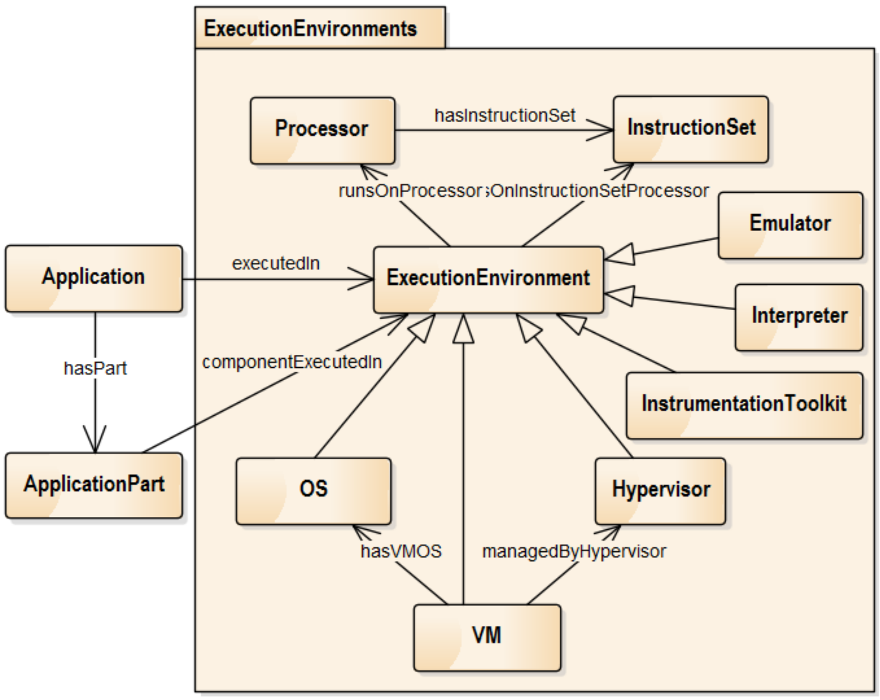

Figure 16: The `ExecutionEnvironments` package

Finally, the `ExecutionEnvironments` package refines the abstract class `ExecutionEnvironment` by subclassing it. Our initial list of the execution environments we consider includes:

- `OS` class, which describes the operating systems where applications may be executed.
- `Processor` and `InstructionSet` classes, which allows us to describe the processing unit for which the application will be compiled.
- `VM`, `Emulator`, `Interpreter`, and `InstrumentationToolkit` classes that describe virtual execution environments. Moreover, we will add more information on Virtual machines as they are managed by one-to-many hypervisors, as described using the `Hypervisor` class instances associated with the `managedByHyperVisor` assocation, and run an operating system, as modeled with the `hasOS` association.

3.4.2 Assets sub-model

Figure 17 reports the categorization of assets presented in D1.02. The different asset categories have been modelled with subclassing. The classes `PublicData`, `TraceableData`, `TraceableCode`, `ApplicationExecution`, `UniqueData`, `GlobalData`, `PrivateData`, and `Code` are subclasses of the main `Asset` class. Moreover, the `Code` class has been further subclassed in `SecurityLibrary`, `CustomAlgorithm`, and `PrivateProtocol`.

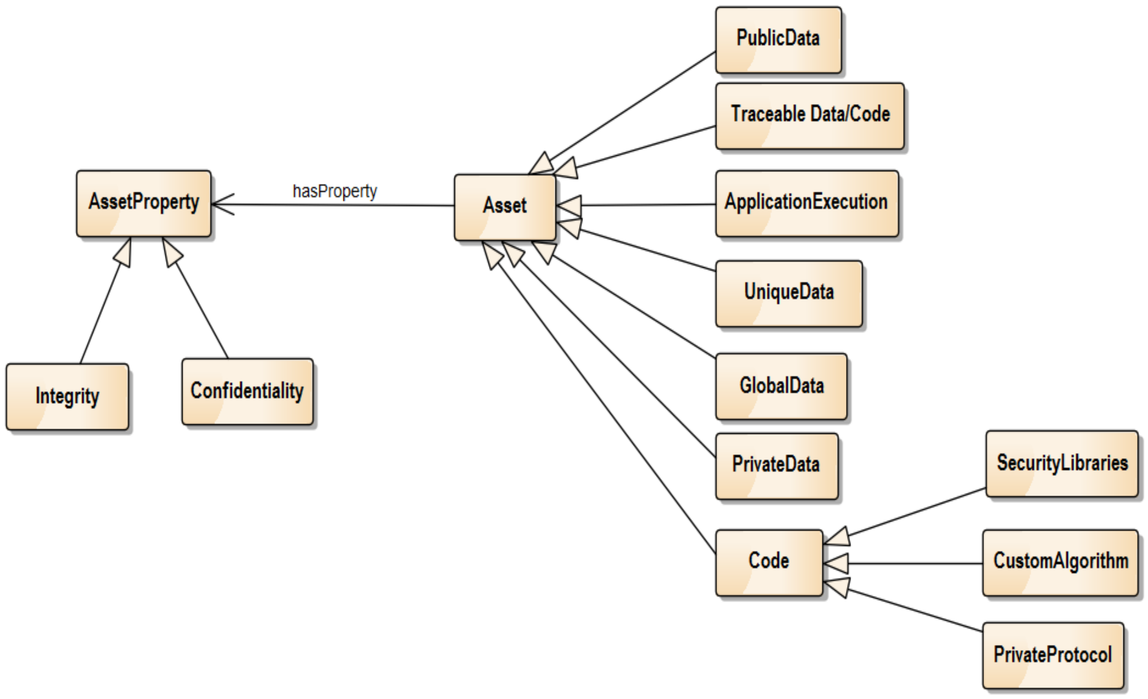

Figure 17: Assets sub-model.

3.4.3 SW Protection sub-model

Figure 18 shows the categorization of the protection techniques that are considered of interest for the ASPIRE project.

First of all, it is possible to see the five lines of defence, i.e., the main, more abstract categories of protections as detailed in the ASPIRE DoW. These techniques are represented by means of the `DataHiding`, `AlgorithmHiding`, `AntiTampering`, `RemoteAttestation`, and `Renewability` classes, all subclasses of the `SWProtectionType` class. Together with the five lines of defence, we also added some more concrete protection types that play a major role in the ASPIRE project.

They are represented by `ClientSideCodeSplitting`, `ClientServerCodeSplitting`, `ClientServerDataSplitting`, and `ReactiveTechnologies`.

All the previously mentioned classes are the first level of categorization. We also specialized some of these techniques, for instance:

- DataHiding has been subclassed in `Source2SourceDataObfuscation` and `WBC` (white box crypto);
- `AlgorithmHiding` has been subclassed in:
  - `SourceLevelAlgorithmHiding`, further subclassed in `PatternRemoval`;
  - `ClientSideVM`;
  - `BinaryCodeObfuscation`, further subclassed in `CodeFlattening`, `BranchFunctions`, and `OpaquePredicates`;
- `AntiTampering`, has been subclassed in `AntiCodeInjection`, `AntiLibraryCallback`, `AntiDebug`, and `CodeGuards`;
- `Renewability` has been subclassed in `RenewabilityInSpace` and `RenewabilityInTime`
- `ReactiveTechnique` has been subclassed in `TimeBombs`.

Integrity protection techniques, like remote attestation and anti-tampering techniques, are also categorized as either static and dynamic. However, instead of using the subclassing we plan to add a Boolean attribute (`dynamic`) in those classes.

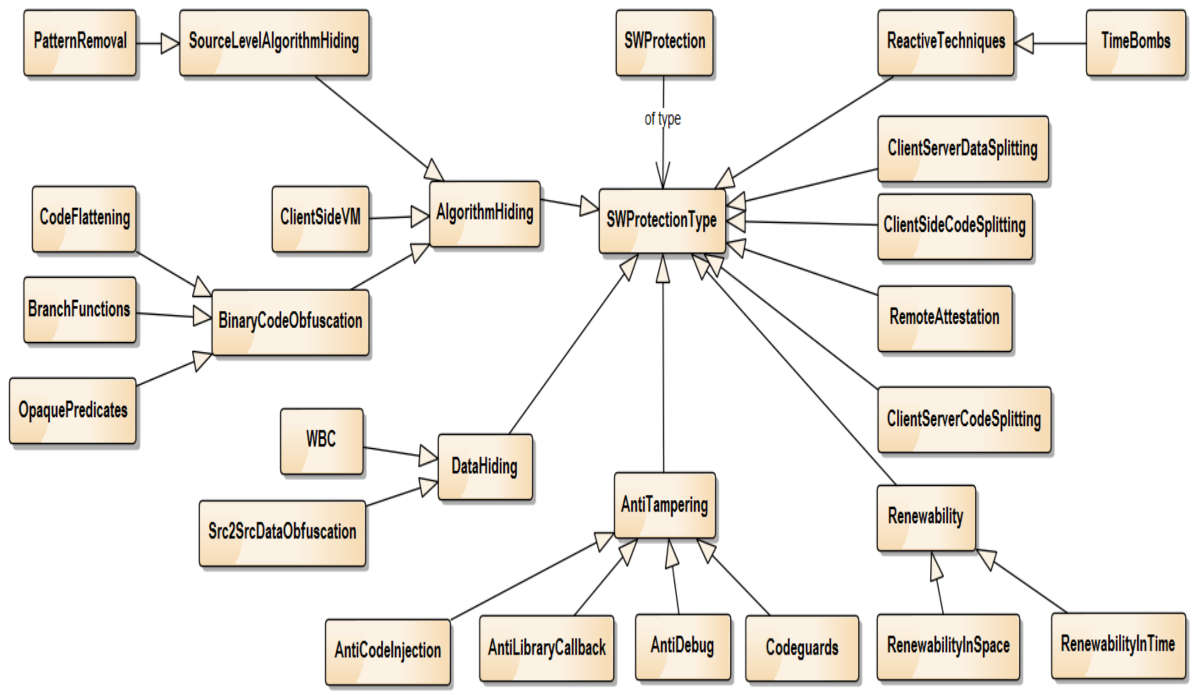

Figure 18: SW Protection sub-model.

3.4.4 Attacks sub-model

Figure 19 presents the attack sub-model. The categorization of the attacks also comes from D1.02, that distinguished static, dynamic, and hybrid attacks. To that extent, we introduced the top-level subclasses `StaticAttack`, `DynamicAttack`, `HybridAttack`, `PassiveAttack`, and `ActiveAttack`.

Static attacks are further subclassed in `StructuralCodeRecovery`, `StructuralDataRecovery`, `StructuralMatchingOfBinaries`, and `StaticTampering`. Dynamic attacks are further subclassed in `DynamicTampering`, `DynamicDataAnalysis`, `Debugging`, `AttacksOnCommunicationChannels`, `DynamicStructuralAnalysis`, and `Fuzzying`. `AttacksOnCommunicationChannels` has been further subclassed in `APIExploitation`, `Spoofing`, `Sniffing`, `Replay`, and `AttacksOnProtectedChannels`.

The Attack sub-model also includes the part of the conceptual model that is needed to represent the Petri nets that will be used for the simulation. Figure 20 depicts this part.

There are two ways to support the simulation with Petri nets: (i) static descriptions of known attack paths, and (ii) dynamic discovery of attack paths from descriptions of attack steps. The latter paths will be generated by the enrichment phases, e.g., when constructing Petri nets for specific scenarios. This capability will allow the developers of the AKB and the ADSS to populate it with generic, widely applicable attack steps that can be reused in many attack and protection scenarios. The former, static descriptions will be useful to manually populate the model with known attacks, e.g., because they are very specific combinations of attack steps that make sense only in a very limited set of scenarios, or, during the project itself, because the automated enrichment and reasoning in the ADSS is not yet mature enough to derive all relevant attack path automatically.

The transitions of a Petri net are instances of the class `AttackStepInAttackPath` - already presented in section 3.3 together with the `refersToAttackStep`, `refersToAttackPath`, and `startsWith` associations, the `Asset`, `Attack` and `AttackPath` classes and associations - which refers to the `AttackStep`. Places of a Petri Net are instances of the class `AttackState`. The relationships `decomposedAND` and `decomposedOR` are needed to further specify dependencies among attack states, that is, it an attack state can be reached if one or more previous states have been reached.

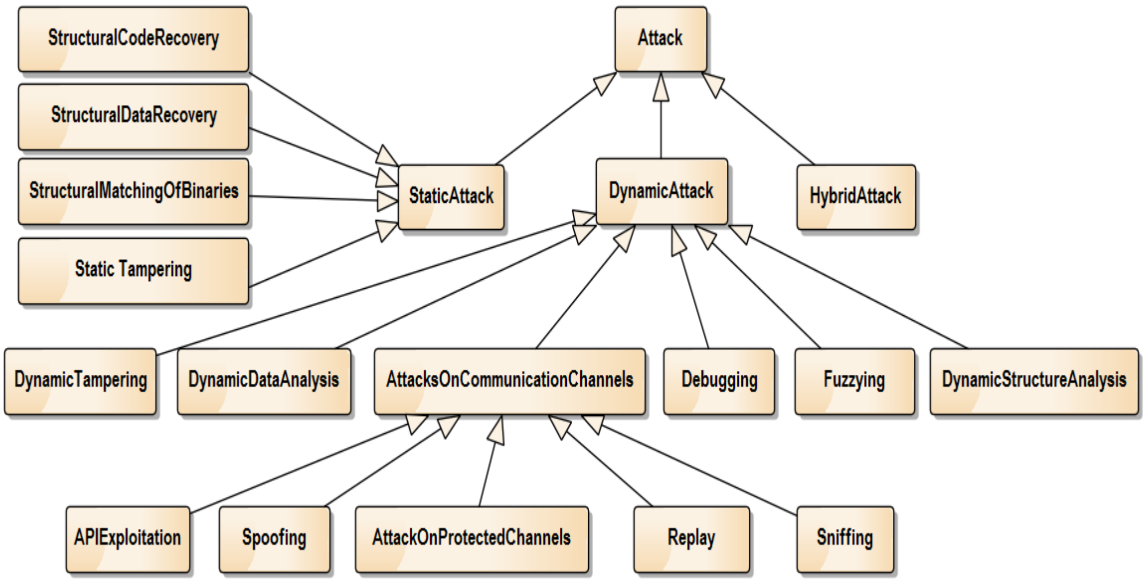

Figure 19: Attacks sub-model

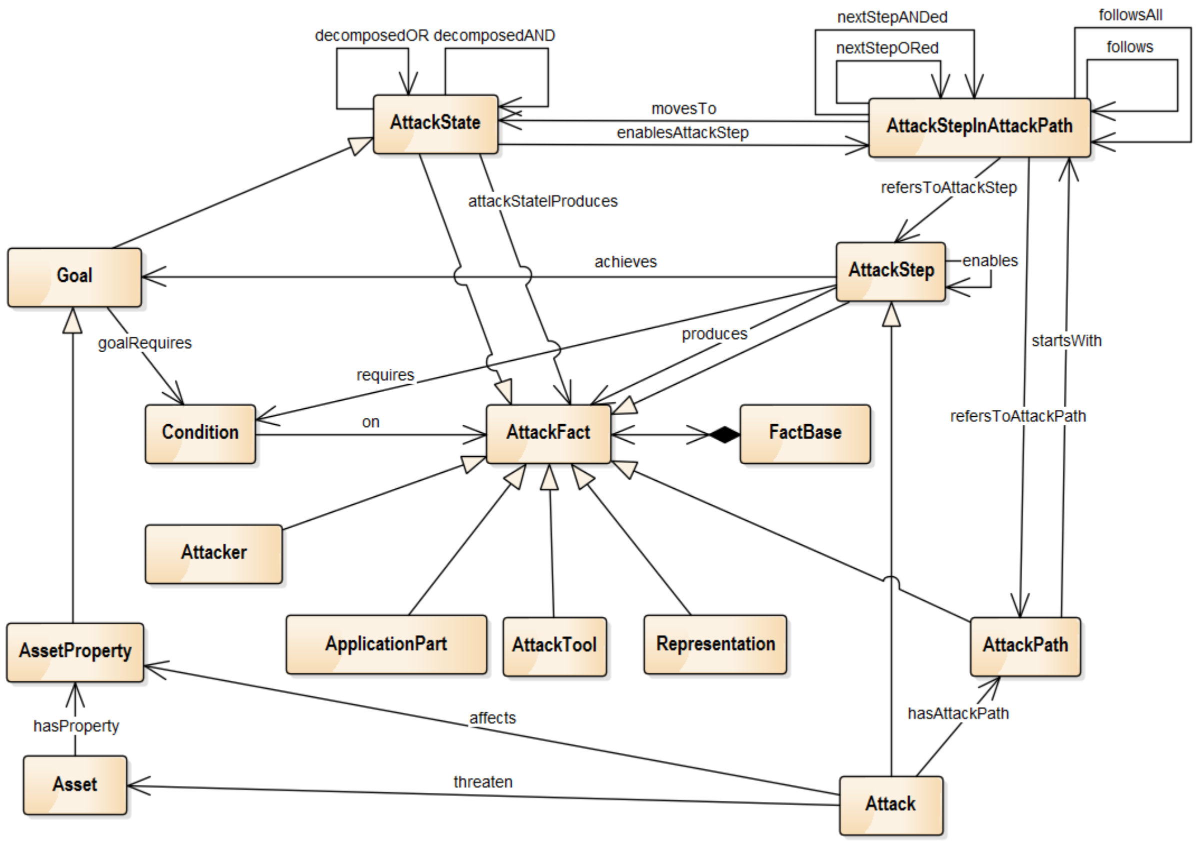

Figure 20: Attacks, goals and Petri Nets

`Goal` is a generalization of `AttackState`, which means that some of them are goals as they contain relevant information on assets or on the state of protections being undone or worked around, like intermediate achievements. Moreover, some of the goals are the final objectives of an attack, i.e., the targeted asset properties. This is represented by defining the `AssetProperty` as a subclass of `Goals`.

Each arc in a Petri net connects one `AttackState` to an `AttackStepInAttackPath` or vice versa; the relationship `movesTo` represents arcs connecting a transition to a place, while the relationship `enablesAttackStep` represents arcs connecting a place to a transition; in this way the whole static structure of a Petri net can be modelled. Starting from the final goal of an attack the model can be navigated through the relationships `movesTo` and `enablesAttackStep` to dynamically discover all the transitions and places leading to this final attack goal.

Together with the `nextStepORed` and `nextStepANDed` associations, we introduced their reciprocal associations to improve querying and reasoning. The relationship `follows` further specifies the fact that one attack step follows another one and can be reached if at least one of the previous attack steps has been completed, and `followsAll` can represent the fact that to perform one attack step more than one previous steps must have been performed.

`AttackState` instances might require some information to be executed, might be performed with an `AttackTool` and will produce some information. Any information used throughout the attack has been named `AttackFact` as the most generic information represented as a fact into a fact base, described by means of the `FactBase` class, composed of `AttackFact` instances.

We can use this information to dynamically discover attack paths. Indeed, `AttackStep` enabling constraints are depicted by (optionally) associating an `AttackState` instance to a `Condition` instance, representing a predicate on some of the facts in the knowledge base, by means of the `attackStateRequires` association between `AttackState` and `Condition` instances, and the on association, between `Condition` and `AttackFact` instances.

The description of the information we expect to use has been obtained by subclassing the `AttackFact` class. Our expectation is to use as facts the `ApplicationPart` instances (as it is important to know what to attack), the associated `Representation` instances (as obtaining `AttackTool` he has at his disposal (as we need to know what the attacker is able to do), and the AttackPaths the attacker likes. Additionally, having performed and completed some `AttackStep` or reached some `AttackState` (and `Goal`) is also important to evaluate if attack steps and states can be executed (and determine dependencies). The known facts are updated when attack steps and attack states are completed. Indeed, we have foreseen two associations to this purpose, `produces` from `AttackStep` instances to `AttackFact` instances, and `attackStateProduces`, from `AttackState` instances to `AttackFact` instances.

In some cases, it is needed to explicit state that an attack step enables other attack steps (without the need to dynamically derive it). This is done by using the `enables` self-association of the class `AttackStep`.

The fact that `Attack` instances are also `AttackStep` instances (by generalization) shows another advantage of the attack submodel: it can manage composition of attack paths. This result will be achieved by hierarchical composition of Petri nets, where one transition (an `AttackStep` instance) can actually represent another sub-net included in the main Petri net.

3.4.5 Metrics sub-model

The project's activity on code metrics in WP4 did not start yet, so the metrics sub-model is just a place-holder for the time being. It will be refined when more details about metrics will be elaborated and validated.

At the moment, this sub-model considers the relations among metrics, application parts, and attacks. The `Metric` class is an abstract class that will be sub-classed when project metrics will be available. Metrics are computed on specific portions of the application. Therefore we added the association `computedOn` between `Metric` instances and `ApplicationPart` instances. The actual values of metrics will be conveyed by means of attributes whose type (integer or real) will be decided depending on the metrics.

Additionally, metrics will be used to quantify the impact of a protection, applied on an application part, against a specific attack. For this purpose, we have foreseen an association, `evaluatesImpactAgainst` between `Metric` instances and `Attack` instances. In this case, we expect to have several attributes as a protection might influence the attack probability, the attack time or expertise required, etc.

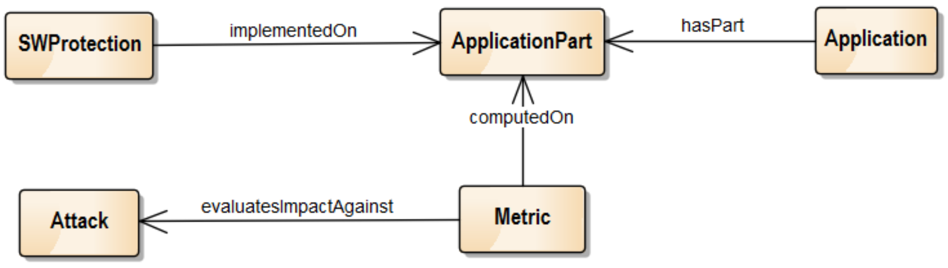

Figure 21: Metrics sub-model

3.4.6 Protection requirements sub-model

One of ASPIRE objectives is to allow an Application Vendor without being an expert on software protection. Therefore, we must allow an ADSS user without an in-depth knowledge on software protection to easily specify his protection requirements. To this purpose, we introduced a set of predefined protection profiles. However, ADSS users can have a strong background on software protection. Therefore, the ADSS must also allow them to fine tune their protection requirements by precisely specifying the attackers they want to face, the attacks they think are relevant, and to restrict the software protection to use. That is, they may want to constrain with precise directives the selection of the best protection for their application.

To describe this scenario, we first introduced a utility concept, the `User` class, which just serves to relate different protection profiles (to different ADSS users). `User` class instances are related via the `requiresProfile` association to the `ProtectionProfile` class instances, the class we introduced to convey information about software protection profiles. Each protection profile is composed of a set of `ProtectionRequirementForAssetProperty` instances that serve to specify the desired protection for a single asset property. The target AssetProperty instance is specified by means of the many-to-one `protectionFor` association.

We initially see two types of protection specifications, thus we subclassed `ProtectionForAssetProperty` in the `SimplifiedProtectionRequirement` and `CustomProtectionRequirement`.

Instances of `SimplifiedProtectionRequirement` can be used to specify the intended hardening level by directly referring to an enumeration that determines the attacker expertise, the `AttackerIdentificationEnum` class. That is, it will be possible for a user to specify that he wants to protect against amateurs, geeks, experts or gurus (see D1.02). The ADSS will first automatically select for the user all the attacks, attack paths and attack tools to protect from, and then investigate the protection techniques to implement.

For a more fine grained specification of the requirements, we introduced the `CustomProtectionRequirement` class, whose instances are composed of a set of

ProtectionTarget instances. ProtectionTarget instances allow users to define a set of “attackers” to protect against. As explained in Section 3.3, the attackers to protect against can be defined as Attacker instances, which are related with their expertise (i.e., with the AttackerIdentificationEnum class), the attack tools at their disposal (i.e., with the AttackerTool class), and the attack path they may be interested in perfoming (i.e., with the AttackPaths class). Therefore, the ProtectionTarget is associated to the Attacker instances via the restrictAttacker association. Morever, a ProtectionTarget instance is also associated to AttackStep instances (via the restrictAttackStepTo) to permit a more fine-grained definition of the weapons available to the attacker. Additionally, a CustomProtectionTarget instance is associated to the SoftwareProtection or SoftwareProtectionTypes instance the user wants to consider during the protections selection phase (via the restrictProtectionsTo and restrictProtectionTypesTo associations).

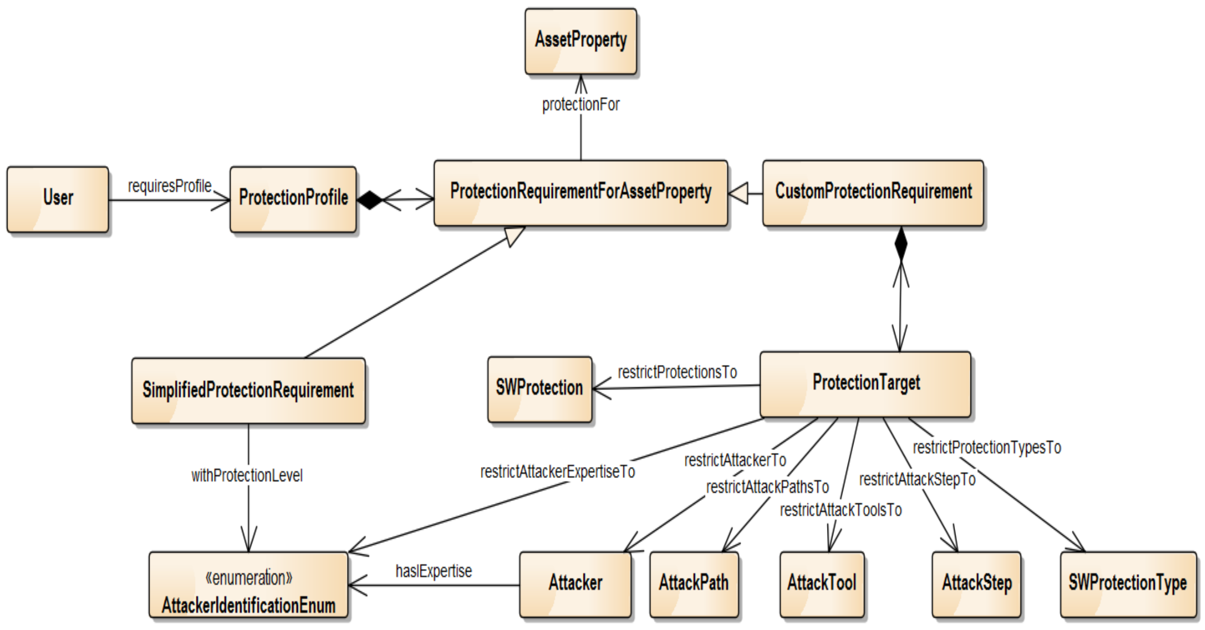

Figure 22: Protection requirements sub-model.

3.5 Mapping a Petri Net in the Knowledge Base

Some of the concepts introduced in our conceptual model (see Fig. 19 in Section 3.4) are used to represent any Petri net in the knowledge base. Here, we define a mapping between the Petri net elements and the classes and relationships defined in the conceptual model.

In order to define such a mapping independently from the language chosen for the knowledge base representation, we decompose Petri nets into triples. We will demonstrate this mapping on the OTP generator example of Section 2.3, of which we replicated the Petri net in Figure 23. In triples of the form <*subject*, *predicate*, *object*> to illustrate how the mapping works: *predicate* can be one of the relationships defined in the conceptual model, while *subject* and *object* are two instances of two classes linked by the predicate relationship; the specialization is represented by the special relationship *isA*, while the instantiation of an object is represented with the special relationship *instanceOf*.

The transitions T0-T11 in this Petri net are objects, i.e., instances of the class AttackStepIntoAttackPath, which refers to the AttackStep, while places P0-P10 are instances of the class AttackState;

Goals are a specialization of `AttackStates`, because some states in an attack correspond to (sub-)goals being reached by an attacker. Examples are P2, when the attacker knows the PIN, and P10, when the attacker has obtained the seed.

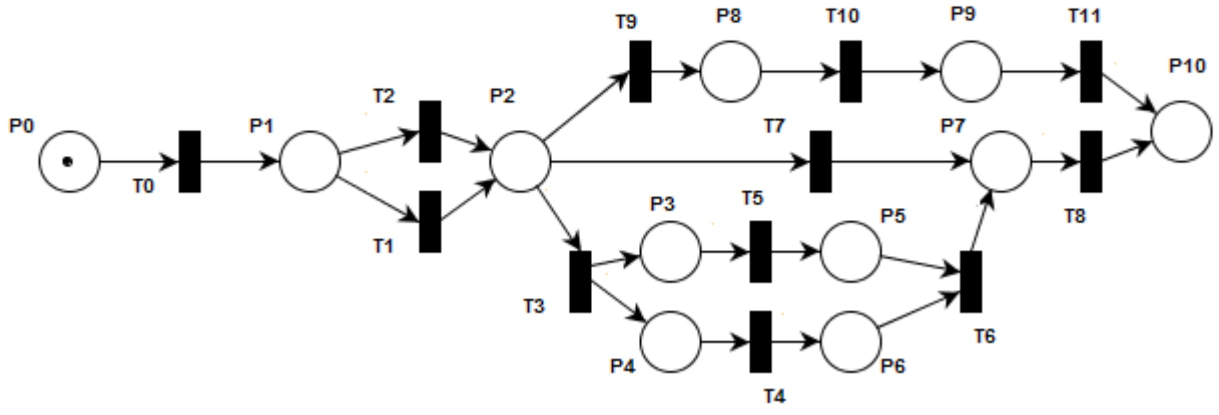

Figure 23: Petri net of the OTP attack (copied from Figure 7).

Each arc in the Petri net, connecting one `AttackState` to an `AttackStep` (and vice versa) must be represented with a triple <subject, predicate, object>, where the predicate can be `movesTo` for arcs connecting a transition to a place, or `enablesAttackStep` for arcs connecting a place to a transition. For example, the sub-net from P0 to P1 is mapped onto the triples of Listing 1.

The choice (OR) pattern represented by P1, T1, T2, and P2, where T1 or T2 can fire to transit from P1 to P2, can be modelled similarly, as depicted in Listing 2. Moreover, the concurrent pattern modelled by T3, P3 and P4 can be mapped as shown in Listing 3, and the AND pattern denoted by nodes P5, P6, T6, P7 can be represented as in Listing 4. Finally, as an attack is composed of different attack paths, it is also possible to model an attack path such as Path 1: {T0, T2, T9, T10, T11}. This is illustrated in Listing 5.

```
T0    instanceOf  AttackStepIntoAttackPath

P0    instanceOf  AttackState

P1    instanceOf  AttackState

P0    enablesAttackStep T0

T0    movesTo P1
```

Listing 1: Representation in triples of the sub-net P0->T0->P1.

```
T1    instanceOf  AttackStepIntoAttackPath

T2    instanceOf  AttackStepIntoAttackPath

P1    instanceOf  AttackState

P2    instanceOf  AttackState

P1    enablesAttackStep T2

P1    enablesAttackStep T1

T2    movesTo P2

T1    movesTo P2
```

Listing 2: Representation of the choice (T1 OR T2) between P1 and P2.

```
T3    instanceOf  AttackStepIntoAttackPath
P3    instanceOf  AttackState
P4    instanceOf  AttackState
T3    movesTo     P3
T3    movesTo     P4
```

Listing 3: Representation of the concurrent pattern (T3 enables P4 and P5 in parallel)

```
T6    instanceOf  AttackStepIntoAttackPath
P5    instanceOf  AttackState
P6    instanceOf  AttackState
P5    enablesAttackStep     T6
P6    enablesAttackStep     T6
T6    movesTo     P7
```

Listing 4: Representation of the AND pattern (P5 and P6 enables T6 to fire to place P7)

```
Path1 instanceOf  AttackPath
Path1 startsWith  T0
T9    instanceOf  AttackStepIntoAttackPath
T10   instanceOf  AttackStepIntoAttackPath
T11   instanceOf  AttackStepIntoAttackPath
T11   follows     T10
T10   follows     T9
T9    follows     T2
T2    follows     T0
T11   movesTo     P10
```

Listing 5: Representation of an attack Path leading to the final goal P10

An `AttackState` is related to the set of information (`AttackFacts`) available to the attacker at that point of the attack. For example, the information (`AttackFact`) used during the OTP attack includes `CodeRepresentation`, like execution traces produced by dynamic analysis attack steps T9 and T11, while T10 (i.e., looking for code portions containing XOR op-codes) will require the execution traces produced by T9 and will produce the lines of the identified code portions.

Moreover, an `AttackStep` might require a (pre)-`Condition` to hold in order to be enabled, and a `Condition` can be seen as an expression using one or more `AttackFacts`.

A `Goal` can be seen as a special `AttackState` where the available information at that point (e.g., the PIN at P2, or the seed at the final state P10) is the actual goal related to an asset targeted by the attacker.

The attack paths can be derived from the final goal P10 by navigating the diagram backwards through the `follows` relationships, thus retrieving all the paths leading to the same final goal.

Section 4 Ontologies and Related Work

Chapter Authors:

Cataldo Basile (POLITO), Daniele Canavese (POLITO), Paolo Falcarin (UEL), Shareeful Islam (UEL), Mariano Ceccato (FBK)

To represent the AKB, we have decided to use ontologies. The complete picture, include all reasons we have for choosing ontologies, will only become available with deliverable D5.01, due in M9, when the ADSS will be presented in details and the full role of the enrichment will be illustrated.

This section introduces some background to formal ontologies, and provides already some hints as to why we choose ontologies. Additionally, we present works on ontology engineering and security modelling related to the ASPIRE knowledge base design and implementation.

4.1 Background on Ontologies

Ontologies are an important tool for the design and implementation of the ASPIRE knowledge base. This section presents a general introduction to the terminology and the main technology available for ontology specification and management. Deliverable D5.01 will report an extended version of this introduction that also covers ontology-based enrichment, formal aspects of description logics, and design best practice for reasoning performance.

The concept of **ontology** comes from the world of philosophy, where it indicates the study of the nature of being and reality in general. Most of the effort in this discipline is focused on answering questions regarding which entities exist or can be said to exist, how such objects can be grouped and how they relate within a hierarchy subdivided according to similarities and differences.

After being adopted by artificial intelligence researchers, in recent years the development of ontology-based tools has begun to surface in actual products and into the industrial world.

In information systems, the main objective of an ontology is the formal representation of the concepts relevant to a specific application domain, according to the (partial or complete) knowledge of the domain. In order to specify a conceptualization, a non-ambiguous definition of the terms representing the available knowledge is needed.

The ontologies define a set of representational primitives that are used to model a specific domain of knowledge or discourse. These primitives are:

- **Individuals** (or *instances* or *objects*) that represent concrete entities.
- **Classes** (or *concepts*) that embody the main categories according to which the world is organized. They are abstract groups, sets and collections of individuals.
- **Properties** (or *roles*) that define a specific relationship between pairs of individuals in the represented model.

Figure 24 graphically presents a sample of the application of the aforementioned entities using a Venn-like diagram.

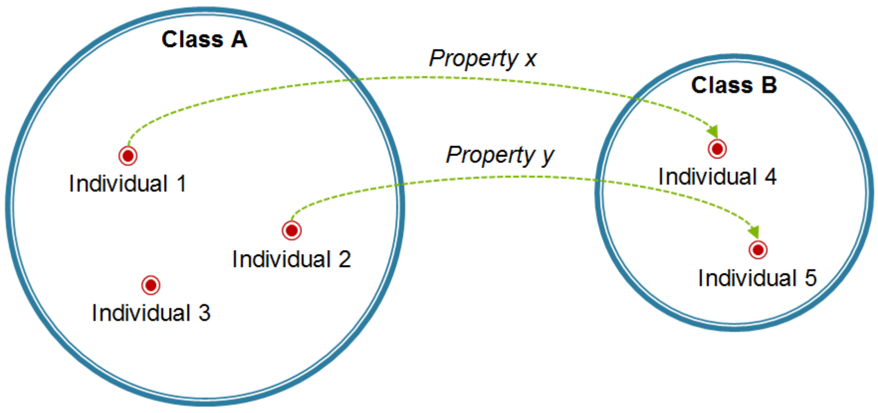

Figure 24: Graphical representation of some ontological primitives

Furthermore, ontologies offer several inference computation capabilities, a.k.a. *reasoning*, that are performed by a specific component called *reasoner*. The Description Logic (DL) ontologies we will use in ASPIRE, are built on top of a well know family of logics, the description logic, which is a decidable part of the first order logic (FOL), and which hence ensures a great expressiveness, computational completeness, and decidability.

Description logic reasoning and queries can rely on highly optimized reasoners that can deliver higher performances than traditional rule engines. Some of the most used and powerful reasoners are:

- HermitT (http://www.hermit-reasoner.com/);
- Pellet (http://clarkparsia.com/pellet/);
- FaCT++ (http://owl.man.ac.uk/factplusplus/).

4.1.1 OWL

From a development point of view, ontologies are implemented using formal languages. These languages are typically declarative in nature, which means that the ontology is specified without providing an algorithm that describes how it should be constructed or how the inference computation should happen. Examples of ontology languages are Common logic [COM], RIF [RIF], and most notably OWL [OWL], which aims to be the standardized and broadly accepted ontology language.

The *Web Ontology Language* (OWL) is built on top of the *Resource Description Framework* (RDF) [RDF] and RDF Schema [RDFS]. RDF provides a way to express statements about resources (that is, about anything) in the form of subject-predicate-object expressions or triples, like the ones already introduced in Section 3.5 to describe attacks. RDFS is a semantic extension of RDF that provides mechanisms for describing characteristics of resources (such as the domains and ranges of properties), groups of related resources, and the relationships between these resources. In practice, RFDS adds a meta-layer that provides a mechanism to describe RDF resources, i.e., to support classes, properties, values, and associations, in a way that is close to the paradigm of object-orientation.

OWL is an extension of these languages that provides superior machine interpretability. It allows one to specify

- logical expressions via the traditional operators (AND, OR and NOT);
- equalities and inequalities;
- local properties;
- required and optional properties;
- required values;
- enumerated classes;
- symmetric and inverse properties.

OWL defines three increasingly expressive sub-languages:

- **OWL-Lite** can be used to express taxonomies with simple constraints such as the cardinality constraint where only the values 0 and 1 are allowed;
- **OWL-DL** combines maximum expressiveness with computational completeness and decidability because it is based on Description Logic[4];
- **OWL-Full** supports full expressiveness and syntactic freedom. This comes at the cost of having no computational guarantees, and hence being semi-decidable. Therefore, the inference computation is less predictable and it is not possible to perform a completely automated reasoning.

OWL 2 is the successor of the Web Ontology Language (OWL 1). Backward compatibility is ensured. All OWL 1 ontologies are completely valid OWL 2 ontologies, but not vice-versa since the latter introduces a number of new and useful set of features such as

- keys;
- property chains;
- richer data types and data ranges;
- qualified cardinality restrictions;
- asymmetric, reflexive and disjoint properties;
- enhanced annotation capabilities.

The OWL 2 standard declares two sub-languages: **OWL 2 DL** and **OWL 2 Full**, which are the obvious extensions of OWL-DL and OWL-Full. In addition, it defines three OWL 2 DL sub-languages (or profiles) which can provide important advantages in certain application scenarios. Each of them is a syntactic restriction of the OWL 2 structural specification and is computationally complete. These profiles are the following:

- **OWL 2 EL** supports polynomial-time algorithms for standard reasoning tasks and it is particularly useful in situations with very large ontologies;
- **OWL 2 QL** enables conjunctive queries to be answered in logarithmic space using standard relational database technology;
- **OWL 2 RL** allows the implementation of polynomial-time reasoning algorithms using rule-extended database technologies operating directly on RDF triples. It is useful in environments with large number of individuals and where it is necessary to operate directly on RDF triples.

Figure 25 shows the expressivity relationships between the OWL 2 profiles. Note that the OWL 2 DL profiles are intersecting since they share some common operators.

4.1.2 SPARQL and SPARQL-DL

SPARQL is a query language for RDF datasets, but it can also be applied to ontologies since every OWL ontology is an RDF dataset [SPARQ].

Like SQL for relational databases, SPARQL can be used to retrieve a sub-set of the data in a dataset. It is able to add and delete information, as well as to modify it. Moreover, SPARQL is capable of federated queries, i.e., retrieving and combining data from multiple datasets available from multiple servers. It can even be used to federate data from non-RDF sources, since there are many tools which make existing formats, such as databases, spreadsheets, and XML files, available to be “sparqled”.

[4] Description Logic is a family of formal logics used for artificial intelligence because it is expressive enough to cope with knowledge representation but guarantees good computational performance [BD+03]. For this reason, DL has been selected by the W3C Semantic Web movement and it is used for biomedical information codification.

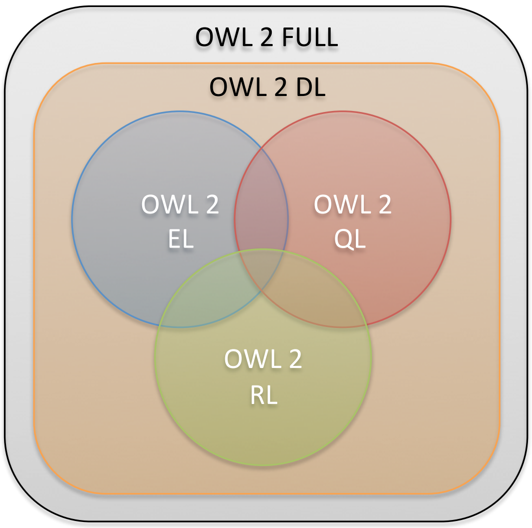

Figure 25: OWL 2 profiles expressivity relationships

SPARQL assumes a client-server interaction model between the program making a query and the processor answering it. For this reason, its specifications consist of three main parts:

- the syntax of the queries;
- the protocol for the communication between the client and the SPARQL processor;
- an XML format for the results of a query.

SPARQL has been designed to be the analogous, in the ontology world, of SQL for databases, as suggested by the name. The features of this language for searching data are similar to the SQL's ones. It enables filtering, sorting, aggregating, counting, grouping, performing calculations and comparisons. However, it cannot be considered a really equivalent language, as not all the SQL instructions are available in SPARQL, and vice versa. Furthermore, the biggest difference is in the underlying model: Relational algebra for SQL is substituted by a *triple pattern* approach, where information is linked for query purposes by a subject-verb-object approach to build *graph patterns*, to be matched against the RDF dataset (instead of using the join operation).

For instance, the trivial example shown in Listing 6 returns a set of pairs composed of a person (`?x`) and his uncle (`?z`) obtained by creating a graph pattern having as nodes the individuals `?x`, `?y`, `?z`, and as edges the object properties `hasFather` and `hasBrother`. The listing clearly shows the typical usage of RDF triples (subject-predicate-object) in the code.

```
PREFIX p: <http://someurl/>

SELECT ?x ?z
WHERE
{
    ?x p:hasFather ?y .
    ?y p:hasBrother ?z .
}
```

Listing 6: A trivial SPARQL query

SPARQL-DL is a sub-set of SPARQL specifically tailored for working with OWL-DL and OWL 2 DL ontologies [Sir07]. It has a simpler syntax, it is less expressive (for instance, SPARQL supports regular expressions while SPARQL-DL does not) and allows only two types of instructions:

- **select queries**, which are used to retrieve some data from an OWL ontology (as their SQL counterpart);
- **ask queries**, which are Boolean queries used to test the absence or presence of a particular entity in a knowledge base.

For example, Listing 7 shows a simple SPARQL-DL query that can be used to retrieve the complete class hierarchy by means of direct class and sub-class pairs. Note that the WHERE part does not use triples anymore.

```
SELECT ?c ?d
WHERE
{
    DirectSubClassOf(?c, ?d)
}
```

Listing 7: A simple SPARQL-DL query

4.1.3 SWRL

Rule inference reasoning is widely used in knowledge management systems. Recently some combinations of theorem proving systems and rule inference systems have been proposed to address some limitations of decidable theorem proving systems. One of the main advantages of combining rules and classical theorem proving systems is the support for *complex property chains*. Property chaining is the ability to define a role as the composition of other roles. For this purpose, W3C proposed the *Semantic Web Rule Language* (SWRL) as the standard for integrating rule-based deductions into systems that represent knowledge as a set of RDF triples such as OWL 1 and 2 [SWRL].

The SWRL language defines the syntax for expressing rules to declare property chains in a Horn-like form [OKT05]. For example, Listing 8 shows a simple rule which can be used to detect the “uncles” in an ontology containing a family-like taxonomy.

```
hasFather(?x, ?y), hasBrother(?y, ?z) -> hasUncle(?x, ?z)
```

Listing 8: A simple SWRL rule

The chaining of the `hasUncle` property is graphically depicted in Figure 26.

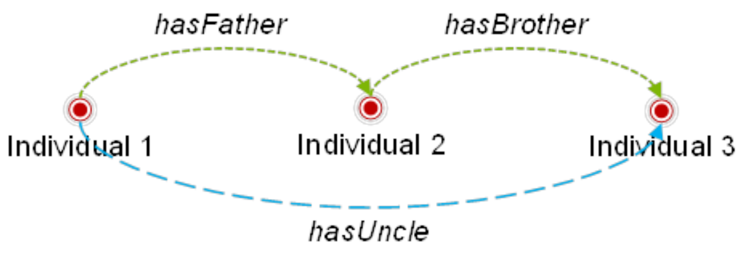

Figure 26: Representation of the hasUncle property chaining

In order to preserve the sound joint system decidability and computation completeness ontologies guarantee, SWRL specifies a set of constraints. The most important one is that SWRL rules cannot be used to add or remove classes, properties and individuals. That is, SWRL can assign couples of existing individuals to an existing object property (named *property assertion*) like stating that Individual1 and Individual2 are bound by the hasUncle object property. However, it does allow neither the creation of the hasUncle property nor the creation of the Individual1 and Individual2 individuals.

It is worth noting that OWL 2 introduced a specific syntax for property chaining. However, its expressiveness is insufficient to express complex topological relationships. A notable example of a problem that can be addressed with SWRL but not with OWL 2 schema level chaining is the detection of object property chains.

4.2 Ontology Engineering

Modelling a knowledge base is a critical task in several research fields and real world applications such as expert systems. A formal representation of the information is a necessity not only to avoid ambiguities, but also to automatically infer features and properties about a system. In order to develop such models (and meta-models), a great variety of logic systems can be used, but when flexibility comes into play, ontologies are frequently one of the most promising approaches. In the security area, there are several ontology-centric works and proposals with a varying level of granularity and applicability.

Souag et al. analyzed several security ontologies, both IT and non-IT specific [Sou12]. They classified them in eight families based on the specific security aspect that they are able to represent and their domain of interest, such as the web oriented and risk based ontologies. They also evaluated the modeling and semantic capabilities of such taxonomies in a semi-qualitative way. Their effort in categorizing ontologies will be considered when precisely defining our ASPIRE ontology.

Avizienis et al. proposed a classification for several security related concepts such as dependability, integrity and confidentiality [Avi04]. They only proposed a taxonomy without giving a formal definition of inference rules and axioms. Their work is purely theoretical, but can be easily implemented in an ontology. It is an interesting example that can lead the design of the ASPIRE security properties, but it is not directly usable in ASPIRE, where we need to actually develop a formal ontology.

Karyda et al. suggested an ontological system for e-government applications (such as e-tax and e-voting) [Kar06]. They provided an OWL ontology which can describe the traditional security concepts (assets, threats and vulnerabilities) and validated their system by performing a set of nRQL (new Racer Query Languagequeries in order to prove its flexibility and adaptability in several different scenarios) [WM05]. nRQL has been designed specifically for analytics that allows time-based data aggregation and filtering. Their model has been already considered when designing the ASPIRE main model. Unfortunately, given the different scenario and especially, given that they use the concept of vulnerabilities we explicitly excluded, we cannot reuse their diagram. However, use of nRQL will be considered in ASPIRE as an efficient way to aggregate and filter ontology data.

Tsoumas et al. described an ontology-centric framework for acquiring and managing security related data [Tso06]. The core ontology is modeled in OWL and it is inspired by the Distributed Management Task Force (DMTF) standard Common Information Model (CIM). Their approach allows not only to represent assets, countermeasures and risks, but also to specify several IT related notions such as network protocols, hardware and software component. IT notions from this work will be evaluated when defining the application sub-model and their packages, however, for the same motivations as with the ontological system of Karyda et al., we cannot reuse their threat model in ASPIRE.

Another interesting work in the area was given in 2007 by Herzog et al. [Her07]. They proposed a generic but flexible core ontology for modeling threats, vulnerabilities and countermeasures for the IT world. In addition they also presented a number of ad-hoc sub-ontologies for several specialized domains such as source code analysis and memory protection sub-ontologies[5]. Their approach on the definition of ad-hoc sub-ontologies for several specialized domains has been used during the design of the ASPIRE Knowledge

[5] All the OWL files are freely available online at http://www.ida.liu.se/~iislab/projects/secont/.

Base and we will certainly consider their specialized ontologies for source code analysis and memory protection.

Assali et al. described how to represent and perform risk analysis in an industrial scenario using several ad-hoc taxonomies and inference systems [Ass08]. They proposed a system consisting of an ontology-based core that can be queried via a web interface for analysing the safety-level of the system. Also in this case, the approach is interesting for our purposes but the reusability of their models is not possible in ASPIRE.

One of the most prominent and renowned works in this field is due to Fenz et al. [Fen09a,Fen09b]. In these documents, they described a set of ontologies that can be used to model the security knowledge of a system by incorporating the most relevant information such as threats, vulnerabilities, assets, controls and their relationships. The proposed ontologies were created by using as a guideline a number of security standards such as the IT Grundschutz Manual [ITG03] and the ISO/IEC 27001 [ISO27k]. The taxonomies are not IT specific and can be used to model any system in which safety is a necessity. Recently, Fenz et al. described FORISK, an expert system for semi-automatically infer the security controls needed to protect a system using the aforementioned ontological systems [Fen13]. The FORISK architecture will be considered when designing our enrichment framework and their core classes (vulnerability excluded) have been inspiring us when defining the main model of the ASPIRE knowledge base.

Massacci et al. proposed an ontology for formally describing security requirements using high-level concepts that underlie natural language and human cognition, such as propositions, goals, and activities [Mas11]. Furthermore, to prove the flexibility of the system, they presented a case study where they studied the requirements needed in an air traffic control scenario. While an air traffic control scenario is far from ASPIRE scope, the idea of modelling requirements as a set of related propositions, goals, and activities will be considered for the protection requirements sub-models.

4.3 Use of Ontologies in ASPIRE

In ASPIRE, we will use DL ontologies to be represented with OWL 2 DL. Being based on description logics, it is possible to automatically perform logical inference to perform sophisticated tasks, like checking the ontology for inconsistencies. Additionally, DL ontologies are endowed with plenty of tools to manage, write, query, and reason about knowledge represented in the ontology; all the “DL” versions of these tools offer better performance and more guarantees (e.g., on decidability) than the general ones.

In this section, we argue that ontologies are appropriate for the purposes of this deliverable, that is, to present the (static) conceptual model of the ASPIRE knowledge base. While the fact that ontologies are able to represent class diagrams is easy to accept, we want to show that the side effects of the use of ontologies are easy to deal with.

The main features required to represent our conceptual model are expressiveness and extensibility.

For what concerns **expressiveness**, ontologies are able to represent classes, class instances (called individuals), associations (called object properties), and attributes (called data properties). That is, they can model class diagrams and entity-relationship diagrams like the ones presented in Section 3.

However, there are differences that need to be remarked, compared to object-oriented or E/R approaches. When used with full expressiveness (that is, in case of OWL Full ontologies), classes are also instances, which makes it possible to represent relations between classes, not only between class instances. This feature allows class-level reasoning, which can be useful in ASPIRE to determine relations between protections. However, given the unavailability of an ontology-based reasoner able to deal with OWL Full and the impossibility of guaranteeing decidability in case of class-based reasoning, even if some OWL Full feature could be of interest for the design of the ASPIRE knowledge base, we will use DL ontologies.

Additionally, ontologies use the Open World Assumption (OWA) whereas object-oriented and entity-relationships [Che02] approaches use the Closed World Assumption (CWA). If a formal system (e.g., a formal logic) follows the CWA, it is implicitly assumed that any statement is considered false, unless it is not possible to explicitly declare its value as true by means of an explicit assignment or by logical inference. With OWA, any statement is assumed as true (resp. false), only if it is possible to explicitly declare its value as true (resp. false). That is, the truth values of any statement are decoupled from any implicit assumption.

The differences between CWA and OWA are important from the philosophical/theoretical point of view: OWA is preferred when no entity knows the entire system hence no entity can make any assumption on the truth value of any statement. When there is such an entity, CWA is indeed to be preferred. But there is also a major difference in practice. Statements for which there is no explicit information on their truth value are assumed as unknown with OWA and false with CWA. Therefore, when designing the inferences needed by the ADSS to determine the combinations of protections to apply on a target application, the fact that ontologies use OWA is one of the first aspects we are considering.

Ontologies must precisely depict the knowledge about a specific domain, but first, domain knowledge can be refined, second, it could be needed to combine the knowledge of more domains. Therefore, ontologies have been designed with **extensibility** in mind. Their extensibility features are enough for our purposes.

There are two standard techniques that can be used to achieve such a result: ontology inclusion and merging, both graphically depicted in Figure 27. Ontology inclusion works very similarly to the `#include` preprocessor directive in C/C++ and it is implemented in the OWL 2 standard. The host ontology explicitly specifies the sub-ontologies to include. When the main ontology is loaded, the system automatically looks-up the sub-ontologies on the file system or on the web, after which they are joined. The ontology merging method is very similar to inclusion. During the merging process, the sub-ontologies axioms (i.e., roughly speaking, everything we know in one domain, including classes, individuals, data/object properties and inference rules) are directly copied into the main ontology using external tools. From a reasoning point of view there is no difference between the aforementioned techniques. However, merging has the advantage to generate a monolithic ontology which subsequently does not require any of the originating sub-ontologies. In both cases we have the same representation of the knowledge domain and approximately the same size and computational performance (the “sum” of the “inclusion ontology” with the included ones is usually very slightly bigger than the merged ontology as the inclusion ontology also stores links to the original ontologies). Currently, we did not make a decision on the method to use for the ASPIRE knowledge base. Most likely, we will create a stable core part of the knowledge base with merging thus we will use inclusion for parts that need to be dynamically joined.

Performance on accessing and querying ontologies is not comparable to the performance of traditional DB. Additionally, ontologies are not very compact to represent, as OWL-DL is an XML-based format. However, having a performance that is not really superb for most of the activities is the natural price to pay for using such a powerful tool. Fortunately, not having satisfactory performance is the only risk we see at the moment, as for all the other aspects ontologies defeat the other traditional modelling approaches.

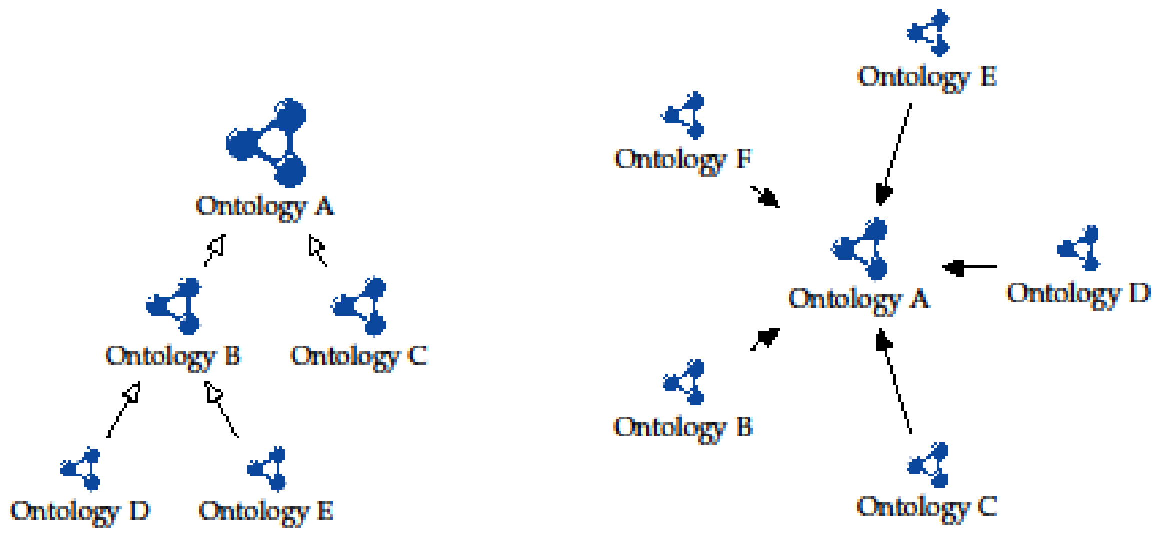

Figure 27 : Examples of ontology inclusion (left) and ontology merging (right).

Nevertheless, as ASPIRE is a research project we are convinced that investigating the full potential of ontologies is worth the risk of not having excellent performance. In fact, we plan to investigate and evaluate other implementation and optimization tricks during the third project year in case the performance would not be fully satisfactory. We are currently investigating several approaches, some are sophisticated (and research intensive), other are more implementation tricks. Furthermore, there already exists literature on best practice for ontology design to improve reasoning performance [LS08,H14]. The problem is that reasoners are based on different techniques. Hence what is good for one reasoner (e.g., for Pellet which uses tableaux reasoning) is not necessary good for another (e.g., Hermit which uses hyper-tableaux). In case of performance issues for the DL reasoning, we will investigate an 'adaptive merging approach', that is, we will only join the axioms from the required sub-ontologies, in order to optimize the reasoning process techniques. At the implementation level, we are considering the use of XML databases to store data and to derive the full ontology or we could maintain more than one synchronized format, one for I/O and querying and one for reasoning. However, the use of equivalent representations will be decided later, when, depending on the assessment of the types of operations needed by the ADSS, we will balance the costs to maintain more synchronized representations with other computational advantages.

To conclude, since there is not a unique optimal approach to address performance issues, the actual optimization will be done later, on the final knowledge base due at M24.

4.4 Use of Ontologies in other EU Projects

Ontologies are seeing a practical use that is constantly growing both in several research areas and actual implementations. One prominent and interesting usage can be seen in the PoSecCo FP7 European Project (http://posecco.eu/), a recently and successfully concluded international project about the management of policy-based systems. The main aim of PoSecCo was to ease the job of the administrators and IT experts by giving them a set of powerful tools able to manage business requirements, policies and low-level configurations of networks and distributed systems with a very minimal human intervention.

In this context, a powerful toolbox named SDSS (Security Decision Support System) was implemented by making an extensive use of ontologies and inferential engines [Pos31,Pos32]. Its main tasks are to:

- edit high-level policies and configurations, if needed;
- harmonize policies (i.e., detect and resolve policy anomalies);

- refine the policies into highly optimized and conflict-free configurations;
- create a report about the conflicts found in the configurations, if manually edited.

A bird's eye overview of the SDSS workflow for refining authorization policies (i.e., policies for configuring firewalls and security protocols such as SSL/TLS) is given in Figure 28.

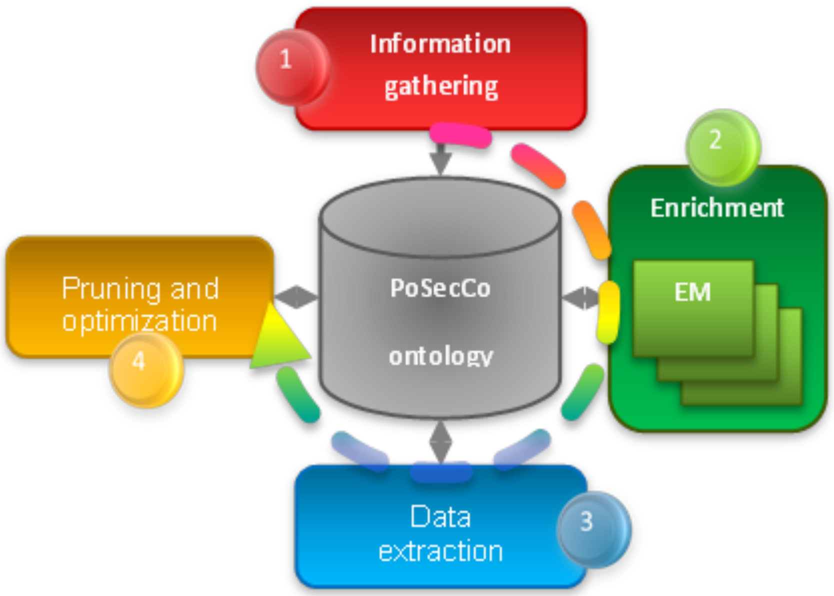

Figure 28: Posecco SDSS workflow

The workflow consists of four consecutive principal phases which make use of a central PoSecCo ontology which acts both as a central repository of all the information required and as a flexible inference-capable tool for detecting additional properties.

The first step is to gather all the information needed for a further analysis about the IT infrastructure to configure. During this phase all the data about the network topology, the nodes' capabilities, the installed software and the policies to refine are collected in order to fill the core ontology. The SDSS is able to read this information both locally from a set of files and also remotely using the data stored in a MoVE server (http://move.q-e.at/).

Once all the available data has been collected, the enrichment phase can begin. The current standard ontological technologies [OWL,SWRL] offers a great level of flexibility, but in a number of practical circumstances their applicability can be severely limited due to constraints imposed by their logic systems. For instance, the open-world assumption can be cumbersome in some scenarios. To overcome these limitations, a common approach is to perform a so-called enrichment of the ontology, where an ad-hoc reasoning is implemented by using traditional programming languages (usually Java) or interfacing the ontology with some external tools (e.g., prolog engines or Boolean satisfiability solvers). These additional inferential components are implemented in a set of software modules called enrichment modules (EMs). The SDSS makes use of a number of EMs for a variety of purposes that includes:

- finding the filtering zones (i.e., the set of nodes which can be traversed without passing through a firewall);
- enabling the WS-Security protocol in a communication if an outsourcing environments is detected;
- identifying the set of not configured web-services.

Once all the data has been collected and reasoned via traditional ontological techniques and custom EMs, the SDSS starts to extract all the possible intermediate representations of the configurations, which are then written back in the ontology for a latter use. These "raw" configurations contain most of the data of the final configurations and are built by performing a set of complex queries on the core ontology, mainly using the SPARQL-DL language, a SQL-like language for interrogating OWL-DL ontologies [Sir07].

The last phase of the refinement consists of two steps. First, an integer linear programming ILP model is dynamically generated and solved in order to detect the “best” intermediate configurations, usually by minimizing the risks or by maximizing the network throughput. Optionally, the user can perform a pruning phase before the optimization begins, in order to discard the least interesting configurations to decrease the computation time of the ILP solving. When the golden intermediate configurations are detected, the final step is to translate them into the final configurations containing all the necessary details for their deployment in the network.

It is a well-known fact that the growing complexity of IT systems is rapidly making them harder and harder to configure and protect. This is certainly true both for networks and software systems. The design of expert systems that can completely substitute the human factor or at least reduce its fallibility is a hot research topic in many areas. The SDSS toolbox has proven that this approach works in a network scenario. The ASPIRE project wants to corroborate the fact that an ontology-based expert system such as the one presented before can be applied successfully in the software protection area. The ADSS will make use of the experience gained during the PoSecCo project and will leverage a similar ontology-centric approach (through enrichment and ILP optimization) to detect and apply the best protection techniques for a software in a semi-autonomous way.

Another project that is making extensive use of ontologies is Multi-Objective Decision Support for Efficient Information Security Safeguard Selection (MOSES3) by SBA Research GmbH (actually from TU Wien researchers), funded by the Austrian Science Fund (FWF) [Mos13,Mos14]. The MOSES3 project focuses on describing attacks and countermeasures against network-based applications in corporate scenario, i.e., traditional network security. MOSES3 proposes the use of a dynamic attack model, where adversaries are modelled as agents that make active, probabilistic decisions in attacking a system, deliberately exploiting dynamic interactions of vulnerabilities. The knowledge gained with this attack model is then used to enable decision-makers to evaluate how to improve the “security” by enforcing remediation and mitigation strategies.

MOSES3 bases all its results on the use of a knowledge base, which mainly includes attacks, countermeasures and a precise definition of the target system (in their case a network). Attacks are modelled through attack patterns, sets of clauses formed by preconditions and actions. As a consequence of actions, the knowledge base can be updated, e.g., to register a successful attack. In terms of ASPIRE knowledge classification, MOSES3’s attack patterns are part of the a priori knowledge.

System are modelled by a set of facts that must allow the simulator to understand if and when preconditions are met. That is, in MOSES3, systems are described in the execution-specific part of the knowledge, but authors have defined a set of standard predicates (corresponding to our application package classes) used for the preconditions, which are instead in the a priori knowledge.

Probabilistic analysis is performed to simulate all the possible ways attackers have at their disposal to mount an attack. Additionally, attack patterns are associated to standard behaviours to profile adversaries (like we do with our attacker expertise). The analysis is implemented by means of a pool of simulations that execute schedules of discrete events and records various outcome variables which can later be analysed and aggregated. A mathematical framework then tries to capture complex causal relations and timing interactions.

MOSES3’s approach, architecture design, and used technology are certainly inspiring for our work. However, porting their results directly into ASPIRE is impossible, given the completely different environments. We will constantly monitor progress to evaluate which results can be also applied to our software protection scenario.

4.5 Security Modelling: Attack Trees and Petri Nets

Petri nets were first introduced to model sequences of chemical reactions [Pet66], and later utilized to model concurrency and synchronization in distributed systems [Pet77].

Petri-nets are a super-set of state-transition diagrams, and have more consolidated mathematical foundation, and their usefulness for attack modelling was pointed by McDermott as an alternative to attack trees [Der10], as Petri nets are better at representing the actions of simultaneous attackers collaborating on the same attack.

Dalton et al. [Dal06] suggested generalized stochastic Petri nets for attack modelling. Stochastic Petri nets are a type of timed Petri nets where transitions fire after random times; in their work, transition delays were assumed to be exponentially distributed which transformed the stochastic Petri net into an equivalent Markov chain, used to calculate joint probability based on conditional probabilities.

Coloured Petri nets (CPN) have attracted some attention for attacks because they are more expressive than basic Petri nets: in coloured Petri nets, tokens carry data values represented by colour which enables different attackers to be distinguished with separate identities in the model. Wu et al. [Wu08] suggested coloured Petri nets for hierarchical attack modelling: an attack represented at a high level is a simple coloured Petri net where certain transitions actually represent a separate CPN sub-net.

Dahl et al. [Dah06] suggested the use of interval timed coloured Petri nets where tokens carry timestamps and colour: the firing delays of transitions are limited by definite time intervals; their concern is timing-dependent attacks carried out by multiple attackers against different targets.

Wang et al recently proposed to use Petri nets to compute the cost of attacks on protected applications [Wan12]. ASPIRE shares the concepts of attack steps/tasks and attack goals but it aims at better quantifying time and probability of each attack step, while their approach is just ranking the transitions in five levels (from fully automated to fully manual).

Chen et al [Che11] used coloured petri nets to describe attacks in smart grids. As Petri net models require a great amount of manual effort and detailed expertise in security threats, they proposed a hierarchical method to construct large Petri nets from a number of smaller Petri nets that can be created separately by different domain experts: the construction method is facilitated by a proposed model description language that enables identical places in different Petri nets to be matched; their effort might be considered in ASPIRE when modelling and integrating different attack paths sharing the same goal into a unique Petri net.

ASPIRE relies on Petri nets for security analysis, because Petri nets were designed for concurrent processes. They can be utilized for simultaneous actions of multiple attackers more naturally, as multiple attackers can be represented with multiple tokens.

Instead, the attack tree approach for coordinated attacks has major drawbacks because its view is limited to a sequence of actions leading to a final goal, and it cannot easily represent simultaneous actions.

Traditionally attack trees have been the most common type of model for representing known attacks [Sch99, Mau05]. In an attack tree, the root of the tree represents the ultimate goal while the branches show all possible sequences of action steps towards the goal. The modelling approach of an attack tree visualizes an attack as a hierarchy of sub-goals leading to the final goal. Attack trees are a popular modelling approach because they are good at describing an attack in an intuitive visual way; show all attack paths within a broad picture; and can lead to useful mathematical analysis (e.g., risk assessment, vulnerability analysis) if nodes are assigned values. However, attack trees are limited in their view of attacks only arranged in sequential steps, focusing on a single goal and a single attacker.

Attack trees come with different proposals with their own syntax, semantics and tools.

The basic attack tree have been extended to attack graphs with nodes might have associated values or logical “and/or” conditions [She04]. Other proposals of attack graphs have emerged with different semantics and visual representation [Ing09,Gup07,She04, Ou06,San13,Kij10], but they all share the modelling of attack goals as nodes and attack tasks as transitions and their behaviour is similar to state-transition diagrams.

For example, Sheyner proposed system tools for generating and analysing the attack graphs [She04]. They use existing network attack models to generate attack graphs automatically from information on network configuration to analyse the system vulnerabilities [She04b]. Other proposals also utilized attack graphs for modelling network attacks and countermeasures [Ing09,Kij10], while Gupta et al used attack graphs more as a visual aid to document the known security risks of a system, and to capture the possible path attackers could use to attack an application [Gup07]. These related approaches are interesting for our purposes but the reusability of their models is not possible in ASPIRE as they are focused on network security or they lack formal representation.

A formal representation of attack graphs is proposed by Santhanam et al. who provided a framework for analysing attack graphs [San13], while Jha et al. utilized model-checkers to analyse attack graphs [Jha02]. They share the same high-level goal of ASPIRE but they use different technologies (attack graphs and model-checkers) to analyse threats in a different domain (network security).

The following works might also be important for ASPIRE, even if they use models different from Petri nets: Ou et al. described attack graph generation from a PROLOG knowledge base [Ou06]. As the AKB will be defined with OWL instead of PROLOG, we will get useful insights from this work on how to adapt it to extract attack paths from the ASPIRE knowledge base. Roy et al. [Roy12] proposes Attack countermeasure trees (ACT) to extend attack trees to take into account both attacks and protections; Xu et al. [Xu06] also models attack with petri nets and they define aspect-oriented petri nets to superimpose protections as sub-nets to be interconnected with the attack model; the way in which the connections among attacks and protections are modelled in such approaches can be adapted to the ASPIRE attacks modelling and influence the knowledge base structure. Bistarelli et al. [Bis07] identifies the best configuration of defense strategies to protect a system modelled with CP-nets and defence trees.

Xie et al. used Bayesian networks for security analysis under uncertainty [Xie10], while Piétre-Cambacédès et al. used Markov processes to evaluate attacks probability [Cam10]. ASPIRE can leverage on both approaches to adapt them to evaluate the probability of attacks.

Braynov et al. [Bra03] proposed a coordinated-attack graph, where nodes represent system states and arcs depict actions causing state transitions: a transition can be executed only if its preconditions are true, and its execution creates some post-conditions. They first consider the sequential actions of a single attacker called an individual plan, and then a coordinated-attack plan for a group of attackers can be created by the union of individual plans. A coordinated-attack graph is the union of all coordinated-attack plans that start from the same initial states and reach the same final goal. ASPIRE addresses the same problem of coordinated attacks but our approach is related explicitly to Petri nets, but the ASPIRE innovation is the simulation the attacker's behaviour before and after protections have been applied in order to evaluate protection's effectiveness: in our case, the simulation will involve relevant data related to attacker effort and success probability, collected from code metrics and by doing manual attack experiments.

4.6 Security Modelling

Another set of related works that is worth mentioning comes from the requirements engineering, goal-modelling, and model-based security communities.

The work in model-based security engineering is often based on extensions of the Unified Modelling Language (UML). Jürjens proposes UMLsec as an extension UML, to include the modelling of security related features, such as confidentiality and access control, as well as tool support for UMLsec [Jur04, Jur05].

Basin et al. use UML to specify access control specification in an application and describe how one can then generate access control mechanisms from the specifications [Bas06]. They consider role-based access control and give additional support for specifying authorization constraints for restricting access.

In requirements engineering, Sindre et al. proposed a misuse case driven approach to elicit the software system's security requirements at an early stage of the development of software systems [Sin05]. Visual links are established between use cases and misuse cases that are used to guide the analysis of functional requirements against security requirements and the threat environment. An executable misuse case is proposed by Whittle et al. to specify misuse case scenarios so that potential attacks within a system context and its mitigations are identified and analysed [Whi08]. These works might provide some guidelines when modelling the relationships between attacks and protections in the ASPIRE knowledge base.

Mouratidis and Giorgini introduced SecureTropos as an extension of the Tropos methodology for modelling and analysing system security properties such as confidentiality and integrity requirements [Mou06]. Mouratidis and Giorgini further extended SecureTropos with the notion of security attack scenarios, where possible attackers, their attacks and system resources that can be attacked are modelled. SecureTropos is used to visualize goal models graphically. It has some similarities with attack graphs but it still lacks a formal representation of the models and it is more focused on security requirement dependencies than on software security as in ASPIRE.

The term Goal Modelling Language mainly refers to the use of goals for analysing system properties and requirements. In particular, Goal-Oriented Requirements Engineering (GORE) has become very popular in recent years for elicitation, evaluation, analysis, and documentation of requirements [Lou95].

i*/Tropos and KAOS are well known methodologies for goal modelling language. i*/Tropos deals with the early requirements, emphasizing the need to understand not only what the organizational goals are, but also how and why the intended system would meet its organizational goals [Yu97, Yu01]. The KAOS (Keep All Objective Satisfied) approach aims to model not only the "what" and "how" aspects of requirements but also "why, who, and when" [Lam09]. A goal in KAOS is a prescriptive or descriptive statement of intent that the system should satisfy through the cooperation of its agents such as humans, devices, and software. The initial goal model in KAOS later on introduced obstacles and constraints that can be seen as boundaries in requirements analysis [Lam00]. Both goals and obstacles can be refined through AND-refinements or OR-refinements for constructing goal model and obstacle model. The goal modelling language is further extended for systematically analyse and manage software development risks.

Such methodologies used to model security requirements relationships might be helpful as guidelines when translating ASPIRE's informal requirements into the concepts and relationships within the ASPIRE knowledge base.

Section 5 Conclusions and Next Steps

This deliverable presented the initial design and concepts of the ASPIRE security model, i.e., the integrated set of modelling techniques that will be used to formally represent and evaluate the different MATE attacks in the ASPIRE project.

The security model represents the core of the overall ASPIRE approach to model and evaluate attacks, as well as the main technologies used in this approach and in the ADSS. The security model defines the simulation models and the structure of the knowledge base, which are the basis of the ASPIRE protection evaluation methodology that will be further developed and extended in WP4, and of the ADSS that will be developed in WP5.

The initial conceptual model in Section 3 is composed of a main model and a set of sub-models which detail and extend the concepts in the main model.

We have already decided to use Petri nets to simulate attacks on vanilla and protected applications, and ontologies to represent the ASPIRE knowledge base. An enrichment framework (incl., e.g., a Petri nets simulator) will be able to dynamically extract the attack paths from the knowledge base, and the output of such simulations will be stored in the knowledge base.

The future enhancements of the security model will be completely described in D4.06 (ASPIRE Security Evaluation Methodology), which will refer to the results obtained throughout the project, as well as in the other deliverables: D4.02 (Preliminary Complexity Metrics), D4.03 (Security Model, Knowledge Base, Human experiments), D4.04 (Security Model, Knowledge Base, Advanced Human experiments), and D4.05 (Public challenge).

In particular, the ASPIRE Security Model and the ASPIRE Knowledge Base will be developed through different phases:

- Formal definition of knowledge base enrichment (M9, D5.01)
- Intermediate update of conceptual model and the attack simulation models for a more in-depth definition of metrics (M12, D4.02);
- Final delivery of the security model (M24 with D4.03 for the conceptual definition, D5.07 for the knowledge base enrichment).

There are many other indispensable aspects of the conceptual model that will only be produced during the next months, such as the ASPIRE tool chain, the characteristics of applications to protect that need to be considered in the security evaluation, the impact of software protections on the asset in the target application, exact relations and effects of the consecutive use of two or more protections on the same application, and so on.

Actual knowledge about concrete attack paths, protections, relations between them, etc. will be described manually by ASPIRE partners and added to the knowledge base as it becomes available as output of WP4 task and result of the ADSS design.

The precise definition of which information will be needed by the ADSS is not fully specified at the time of writing of this report. The definition of the enrichment framework both for the a priori knowledge and for the execution-specific knowledge is therefore foreseen to be delivered in phase 4.

We will therefore elaborate an important sub-model that describes metrics and how they can be computed on vanilla and protected applications. In this document, a very preliminary definition of metrics has been presented. However, since tasks related to metrics (mainly T4.2, but also T4.3, T4.4) did not start yet, results in that sub-model cannot be considered mature. Deliverable D4.02 (M12) will serve to cover this lack in the initial version of the meta-model: D4.02 will report an updated version of the conceptual model that details on metrics (phase 5).

The final version of the ASPIRE knowledge base will be documented in two deliverables: D4.03 (M18) will present the final conceptual model; D5.07 (M24) will present the final enrichment framework.

By taking advantage of the expansibility of both the conceptual model and of the enrichment framework, possible changes that may be required during the last year will be easily supported.

Section 6 List of Abbreviations

Abbreviation	Meaning
ADSS	ASPIRE Decision Support System
AES	Advanced Encryption Standard
AKB	ASPIRE Knowledge Base
API	Application Program Interface
ASM	Attack Simulation Model
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
CFG	Control Flow Graph
CPN	Colored Petri Net
DL	Description Logic
DoW	Description of Work
EMF	Eclipse Modelling Framework
ER	Entity-Relationship
GUI	Graphical User Interface
MATE	Man At The End (attack)
OO	Object-Oriented
OTP	One Time Password
OWL	Web Ontology Language
PIN	Personal Identification Number
PN	Petri Net
PNML	Petri Net Markup Language
RA	Remote Attestation
RDF	Resource Description Framework
RDFS	RDF Schema
SPARQL	SPARQL Protocol and RDF Query Language
SWRL	Semantic Web Rule Language
UML	Unified Modelling Language
WP	Work Package

Bibliography

[Age] AgenaRisk, software for Bayesian Network and Simulation Software for Risk Analysis and Decision Support. On-line at http://www.agenarisk.com

[Anc07] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In Proceedings of the ACM Workshop on Quality of Protection, 2007, pp. 15–20.

[Ass08] Amjad Abou Assali, Dominique Lenne, Bruno Debray, “Ontology Development for Industrial Risk Analysis”, In Proceedings of the International Conference on Information and Communication Technologies: From Theory to Applications (ICTTA 2008), 2008

[Avi04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing”, In Transactions on Dependable and Secure Computing, 2004

[Bas06] Basin DA, Doser J, Lodderstedt T (2006) Model driven security: from UML models to access control infrastructures. ACM Trans Softw Eng Methodol 15(1):39–91

[BD+03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider: The Description Logic Handbook: Theory, Implementation, Applications. Cambridge University Press, Cambridge, UK, 2003.

[Bis07] Bistarelli, Stefano, Fabio Fioravanti, and Pamela Peretti. "Using CP-nets as a guide for countermeasure selection." Proceedings of the 2007 ACM symposium on Applied computing. ACM, 2007.

[Bra03] S. Braynov and M. Jadliwala, “Representation and analysis of coordinated attacks,” in Proc. ACM Workshop Formal Methods Security Engineering (FMSE), 2003, pp. 43–51.

[Bro07] A. Brogi, S. Corfini, S. Iardella. From OWL-S descriptions to Petri nets. In Service-Oriented Computing ICSOC 2007 Workshops - Third international workshop on engineering service-oriented applications: analysis, design and composition (WESOA’07), Wien, Austria, September 17, 2007. LNCS 4907, pages 427-438. Springer.

[Cam10] Piétre-Cambacédès, L., and M. Bouissou. 2010. “Beyond attack trees: dynamic security modeling with Boolean logic driven Markov processes (BDMP)”. In European Dependable Computing Conference (EDCC 2010), 199–208.

[Cap12] A Capiluppi, P. Falcarin, and C. Boldyreff. "Code defactoring: Evaluating the effectiveness of java obfuscations." 19th Working Conference on Reverse Engineering (WCRE), IEEE, 2012.

[Cec08] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, P. Tonella. Towards Experimental Evaluation of Code Obfuscation Techniques. In Proceedings of the ACM Workshop on Quality of Protection, 2008, pp. 39–46.

[Cec09a] M. Ceccato, M. Dalla Preda, J. Nagra, Ch. Collberg, and P. Tonella. Trading-off Security and Performance in Barrier Slicing for Remote Software Entrusting. Automated Software Engineering 16(2), 2009, pp. 235–261.

[Cec09c] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, P. Tonella. The Effectiveness of Source Code Obfuscation: an Experimental

Assessment. In Proceedings of the 17th IEEE International Conference on Program Comprehension, 2009, pp.178–187

[CheO2] P. Chen. 2002. Entity-relationship modeling: historical events, future trends, and lessons learned. In Software pioneers, Manfred Broy and Ernst Denert (Eds.). Springer-Verlag New York, Inc., New York, NY, USA, p. 296-310.

[Che11] Chen, T., Sanchez-Aarnoutse, J.-.C. and Buford, J. (Dec 2011). Petri nets models for cyber-physical attacks in smart grid. IEEE Trans. on Smart Grid, 2, 741-749.

[COL] COLOANE Petri Nets tool. On-line at https://coloane.lip6.fr/

[Col97] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, 1997.

[COM] International Organization for Standardization, "ISO/IEC 24707:2007 - Information technology — Common Logic (CL): a framework for a family of logic-based languages", 2007

[CPN] CPN-tools, Colored Petri Nets tool. On-line at http://cpntools.org/

[Dah06] O. Dahl and S. Wolthusen, "Modeling and execution of complex attack scenarios using interval timed colored Petri nets," in IEEE Int. Workshop Innovative. Architectures Future Generation High-Performance Processing System, 2006, pp. 49–55.

[Dal05] M. Dalla Preda and R. Giacobazzi, "Semantic-based code obfuscation by abstract interpretation," in In Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05).

[Dal06] G. Dalton, R. Mills, J. Colombi, and R. Raines, "Analyzing attack trees using generalized stochastic Petri nets," in Proc. IEEE Workshop Inf. Assurance, West Point, NY, Jun. 2006, pp. 116–123.

[Der10] J. P. McDermott, "Attack net penetration testing," in Proceedings of the 2000 workshop on New security paradigms, New York, NY, USA, 2001, pp. 15-21.

[Dew07] Dewri, Rinku, et al. Optimal security hardening using multi-objective optimization on attack tree models of networks. In: Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007. p. 204-213.

[EPNK] Petri Net editor ePNK; on-line at http://www.imm.dtu.dk/~ekki/projects/ePNK/

[Fal06] Falcarin, Paolo, Riccardo Scandariato, and Mario Baldi. "Remote trust with aspect-oriented programming." Advanced Information Networking and Applications, AINA 2006. Vol. 1. IEEE, 2006.

[Fen09a] Stefan Fenz, Andreas Ekelhart, "Formalizing information security knowledge", In Proceedings of the International Symposium on Information, Computer, and Communications Security (ASIACCS 2009), 2009

[Fen09b] Stefan Fenz, Thomas Pruckner, Arman Manutscheri, "Ontological mapping of information security best-practice guidelines", In Lecture Notes in Business Information Processing, 2009

[Fen13] Stefan Fenz, Thomas Neubauer, Rafael Accorsi, Thomas Koslowski, "FORISK: Formalizing information security risk and compliance management", In Proceedings of the Conference on Dependable Systems and Networks Workshop (DSN-W 2013), 2013

[Gia12] R. Giacobazzi and A. Toppan. On Entropy Measures for Code Obfuscation. In Proceedings of ACM SIGPLAN Software Security and Protection Workshop, 2012, 8 pages.

[Got00] H. Goto, M. Mambo, K. Matsumura, and H. Shizuya. An approach to the objective and quantitative evaluation of tamper-resistant software. In Proceedings Third International Workshop on Information Security, 2000, pp. 82–96.

[Gup07] Gupta, Suvajit, and Joel Winstead. "Using attack graphs to design systems." Security & Privacy, IEEE 5.4 (2007): 80-83.

[H14] K. Hammar, “Reasoning Performance Indicators for Ontology Design Patterns”, presentation http://ontolog.cim3.net/cgi-bin/wiki.pl?ConferenceCall_2014_02_06, 2014.

[Ham11] J. Hamilton and S. Danicic, "An Evaluation of the Resilience of Static Java Bytecode Watermarks Against Distortive Attacks," IAENG International Journal of Computer Science, vol. 38, pp. 1-15, 2011.

[Her07] Almut Herzog, Nahid Shahmehri, Chaudiu Duma, “An Ontology of Information Security”, In International Journal of Information Security and Privacy, 2007

[Hou05] Houmb SH, Georg G, France RB, Bieman JM, Jurjens J (2005) Cost-benefit trade-off analysis using BBN for aspect-oriented risk-driven development. In: Proceedings of the 10th IEEE international conference on engineering of complex computer systems. IEEE Computer Society, pp 195–204

[Hub90] Huber, Peter, Kurt Jensen, and Robert M. Shapiro. "Hierarchies in coloured Petri nets." Advances in Petri Nets 1990. Springer Berlin Heidelberg, 1991. 313-341.

[Ing09] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer, “Modeling modern network attacks and countermeasures using attack graphs,” in Proc. Annu. Comput. Security Appl. Conf. (ACSAC), Austin, TX, Dec. 2009, pp. 117–126.

[Iri10] I. Hadar, T. Kuflik, A. Perini, I. Reinhartz-Berger, F. Ricca, and A. Susi. An empirical study of requirements model understanding: Use Case vs. Tropos models. In Proceedings of the ACM Symposium on Applied Computing, 2010, pp. 2324–2329

[Isl14] S.Islam, H. Mouratidis, and E. Weippl. An Empirical Study on the Implementation and Evaluation of a Goal-driven Software Development Risk Management Model, Journal of Information and Software Technology, Vol 56, Issue 2, February, 2014, Elsevier.

[ISO27k] International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC), ISO/IEC 27001:2013, Information Security Management System (ISMS) standard, 2013

[ITG03] IT Baseline Protection Handbook. Germany. Federal Office for Security in Information Technology. Bundesanzeiger, Cologne 2003-2005.

[Jen09] K. Jensen, L.M. Kristensen, Coloured Petri Nets, Springer-Verlag Berlin Heidelberg 2009

[Jen96] Jensen, Finn V. An introduction to Bayesian networks. Vol. 210. London: UCL press, 1996.

[Jha02] Jha, Somesh, Oleg Sheyner, and Jeannette Wing. "Two formal analyses of attack graphs." Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE. IEEE, 2002.

[Jur04] Jürjens, J.: Secure Systems Development with UML. Springer, Berlin (2004)

[Jur05] Jürjens, J.: Sound methods and effective tools for model-based security engineering with UML. ICSE 2005, ACM, pp. 322–331, (2005)

[Kar06] Maria Karyda, Theodoros Balopoulos, Lazaros Gymnopoulos, Spyros Kokolakis, Costas Lambrinoudakis, Stefanos Gritzalis, Stelios Dritsas, “An ontology for secure e-government applications”, In Proceedings of the International Conference on Availability, Reliability and Security (ARES 2006), 2006

[Kij10] KIJSANAYOTHIN, Phongphun; HEWETT, Rattikorn. Analytical approach to attack graph analysis for network security. In: Availability, Reliability, and Security, 2010. ARES'10 International Conference on. IEEE, 2010. p. 25-32.

[Kin12] J. Kinder. Towards Static Analysis of Virtualization-Obfuscated Binaries. In Proceedings of the 19th IEEE Working Conference on Reverse Engineering, 2012, pp. 61–70.

[Lam09] A. van Lamsweerde. Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, 2009.

[Lam00] A. van Lamsweerde and E. Letier. Handling obstacles in goal oriented requirements engineering. IEEE Transactions on Software Engineering, 26:978–1005, 2000.

[Lin03] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in Proceedings of the 10th ACM conference on Computer and communications security, 2003, pp. 290-299.

[Lou95] P. Loucopoulos and V. Karakostas. System Requirements Engineering. McGraw-Hill, Inc., New York, NY,USA, 1995.

[LS08] H. Lin and E. Sirin, “Pellint - A Performance Lint Tool for Pellet”. 'OWLED' *CEUR Workshop*, 2008.

[Mas11] Fabio Massacci, John Mylopoulos, Federica Paci, Thein Thun Tun, Yijun Yu, “An Extended Ontology for Security Requirements”, In Proceedings of the International Workshop on Information Systems Security Engineering (WISSE 2011), 2011

[Mau05] S. Mauw and M. Oostdijk, Foundations of attack trees, in Proc. 8th Annual Int. Conf. Inf. Security Crypto. (ICISC), Seoul, Korea, Dec. 2005, pp. 186–198.

[Mos13] Kiesling E., Ekelhart A., Grill B., Strauss C., Stummer C. (2013) "A simulation-optimization approach for information security risk management," International Conference on Operations Research (OR 2013), Rotterdam, Netherlands, Sept. 3-6.

[Mos14] E. Kiesling, A. Ekelhart, B. Grill, C. Stummer, and C. Strauss (2014), "Evolving Secure Information Systems through Attack Simulation," 47th Hawaii International Conference on System Science, Hawaii, USA, Jan 6-9.

[Mou07] Mouratidis, H., Giorgini, P.: Secure tropos: a security-oriented extension of the tropos methodology. Int. J. Softw. Eng. Knowl. Eng. (IJSEKE) 17(2), 285–309 (2007)

[Mou07b] Mouratidis, H., Giorgini, P.: Security Attack Testing (SAT)— testing the security of information systems at design time. Inf. Syst. 32(8), 1166–1183 (2007)

[Mou06] Mouratidis,H., Giorgini, P.: Integrating security and software engineering: an introduction. In: Integrating Security and Software Engineering: Advances and Future Actions, pp. 1–14. Idea Publishing Group, Miami, 2006.

[Myl05] G. Myles, C. Collberg, Z. Heidepriem, and A. Navabi, "The evaluation of two software watermarking algorithms," Software: Practice and Experience, vol. 35, pp. 923-938, 2005.

[Net] Netica tool for advanced Bayesian Belief Networks, https://www.norsys.com/

[OKT05] Martin J. O'Connor, Holger Knublauch, Samson W. Tu, Benjamin N. Grosof, Mike Dean, William E. Grosso, and Mark A. Musen. “Supporting Rule System Interoperability on the Semantic Web with SWRL.”, International Semantic Web Conference, volume 3729 of Lecture Notes in Computer Science, page 974-986. Springer, (2005)

[OM] OpenMarkov software tool for probabilistic graphical models (PGMs) http://www.openmarkov.org/

[Ou06] X Ou, WF Boyer, MA McQueen, A scalable approach to attack graph generation, ACM CCS 2006

[OWL] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein, “OWL 2 Web Ontology Language Document Overview”, W3C Recommendation (http://www.w3.org/TR/owl-overview), 2012

[OWL2PN] OWLS2PNMLtool for translating OWL-S service descriptions into Petri nets http://www.di.unipi.it/~brogi/projects/owls2pnml/

[Per09] A. Perini, F. Ricca, and A. Susi. Tool-Supported Requirements Prioritization. Comparing the AHP and CBRank Methods. Information and Software Technology 2009, pp. 1021–1032.

[Pet66] Petri, CA, Communication with automata, 1966. DTIC Research Report AD0630125.

[Pet77] J. L. Peterson, "Petri nets," ACM Computing Surveys (CSUR), vol. 9, pp. 223-252, 1977

[PIP] PIPE2, Platform Independent Petri net Editor 2. On-line at http://pipe2.sourceforge.net/

[PNML] Petri Net Mark-up Language, ISO/IEC 15909 Part 2 standard. On-line at http://www.pnml.org/

[PNW] Petri Net World: on-line at http://www.informatik.uni-hamburg.de/TGI/PetriNets/

[Pos31] PoSecCo consortium, “D3.1 – Initial SDSS Architecture and Workflow”, 2011

[Pos32] PoSecCo consortium, “D3.2 – Security Ontology Definition”, 2011

[RDF] Graham Klyne, Jeremy J. Carroll, “Resource Description Framework (RDF): Concepts and Abstract Syntax”, W3c Recommendation (http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/), 2004

[RDFS] Dan Brickley, R. V. Guha, “RDF Schema 1.1”, W3C Recommendation (http://www.w3.org/TR/rdf-schema/), 2014

[Ric07] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella and M. Ceccato. The Role of Experience and Ability in Comprehension Tasks Supported by UML Stereotypes. In Proceedings of the 29th International Conference on Software Engineering, 2007, pp. 375–384.

[Ric08] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, M. Ceccato and C.A. Visaggio. Are Fit tables really talking? a series of experiments to understand whether Fit tables are useful during evolution tasks. In Proceedings of the 30th International Conference on Software Engineering, 2008, pages 361–370.

[Ric09] F. Ricca, M. Torchiano, M. Di Penta, M. Ceccato and P. Tonella. Using acceptance tests as a support for clarifying requirements: A series of experiments. Information and Software Technology 51(2), 2009, pp. 270–283.

[Ric10] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato. How developers' experience and ability influence web application comprehension tasks supported by UML stereotypes: A series of four experiments. IEEE Transactions on Software Engineering 36(1), 2010, pp. 96–118.

[RIF] Michael Kifer, Harold Boley, “RIF Overview”, W3C Working Group Note (http://www.w3.org/TR/rif-overview/), 2013

[Roy12] Roy, Arpan, Dong Seong Kim, and Kishor S. Trivedi. "Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees." Security and Communication Networks 5.8 (2012): 929-943.

[San13] SANTHANAM, Ganesh Ram; OSTER, Zachary J.; BASU, Samik. Identifying a preferred countermeasure strategy for attack graphs. In: Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop. ACM, 2013. p. 11.

[Sch99] Schneier, Bruce. "Attack trees." Dr. Dobb’s journal 24.12 (1999): 21-29.

[She04a] Sheyner, Oleg Mikhail. Scenario graphs and attack graphs. 2004. PhD Thesis. University of Wisconsin.

[She04b] Sheyner, Oleg; WING, Jeannette. Tools for generating and analyzing attack graphs. In FMCO: Formal methods for components and objects. Springer Berlin Heidelberg, 2004. p. 344-371.

[Sin05] Sindre G, Opdahl AL (2005) Eliciting security requirements with misuse cases. Requir Eng J 10(1):34–44

[Sir07] Evren Sirin, Bijan Parsia, “SPARQL-DL: SPARQL Query for OWL-DL”, In proceedings of OWL Experience and Directions (OWLED 2007), 2007

[Sou12] Souag Amina, Salinesi Camille, Comyn-Wattiau Isabelle, “Ontologies for Security Requirements: A Literature Survey and Classification”, In Proceedings of the Advanced Information Systems Engineering Workshops (CAiSE 2012), 2012

[Sut06] I. Sutherland, G. E. Kalb, A. Blyth, and G. Mulley, "An empirical examination of the reverse engineering process for binary files," Computers & Security, vol. 25, pp. 221-228, 2006.

[SWRL] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, “SWRL: A Semantic Rule Web Rule Language Combining OWL and RuleML”,W3C Member Submission (http://www.w3.org/Submission/SWRL/), 2004

[TINA] TINA, Time Petri Net Analyzer. On-line at http://projects.laas.fr/tina/

[Tso06] Bil Tsoumas, Dimitris Gritzalis, “Towards an ontology-based security management”, In Proceedings of the International Conference on Advanced Information Networking and Applications (AINA 2006), 2006

[Udu05] S. Udupa, S. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In Proceedings of the 12th Working Conference on Reverse Engineering, 2005, pp. 45–54.

[Wan98] Wang, Jiacun. Timed Petri nets: Theory and application. Vol. 39. Dordrecht: Kluwer Academic Publishers, 1998.

[Wan00] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," University of Virginia 2000.

[Wan12] N. Wang, D. Fang, Y.X. Gu, Z. Tang, H. Wang. The Effectiveness Evaluation of Software Protection based on Attack Modeling. In Proceedings of ACM SIGPLAN Software Security and Protection Workshop, 2012, 8 pages.

[Whi08] Whittle J, Wijesekera D, Hartong M (2008) Executable misuse cases for modeling security concerns. In: ICSE ’08: proceedings of the 30th international conference on Software engineering. ACM, New York, pp 121–130

[WM05] M. Wessel and R. Möller. A High Performance Semantic Web Query Answering Engine. In I. Horrocks, U. Sattler, and F. Wolter, editors, In Proceedings of the International Workshop on Description Logics, 2005.

[Woh00] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen. Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers, 2000.

[Wu08] R. Wu, W. Li, and H. Huang, An attack modeling based on hierarchical colored Petri nets, in Proc. IEEE Int. Conf. Comp. Electrical Eng. (ICCEE), 2008, pp. 918–921.

[Xie10] Xie, Peng, Jason H. Li, Xinming Ou, Peng Liu, and Renato Levy. "Using Bayesian networks for cyber security analysis." In Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on, pp. 211-220. IEEE, 2010.

[Xu06] Xu, Dianxiang, and Kendall E. Nygard. "Threat-driven modeling and verification of secure software using aspect-oriented Petri nets." Software Engineering, IEEE Transactions on 32.4 (2006): 265-278.

[Yu97] E. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In RE ’97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, page 226, Washington, DC, USA, 1997. IEEE Computer Society.

[Yu01] E. Yu, L. Liu, and Y. Li. Modelling strategic actor relationships to support intellectual property management. In ER ’01: Proceedings of the 20th International Conference on Conceptual Modeling, pages 164– 178, London, UK, 2001. Springer-Verlag.

[Zha12] Y.-J. Zhao, Z.-Y. Tang, N. Wang, D.-Y. Fang, and Y.-X. Gu, "Evaluation of code obfuscating transformation," Ruanjian Xuebao/Journal of Software, vol. 23, pp. 700-711, 2012.