Advanced Software Protection:
Integration, Research and Exploitation

D4.03
Security Model, Knowledge Base, Human Experiments

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	November 1, 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10
Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D4.03
WP and tasks contributing:	WP 4 / Task 4.3
Due date:	Oct 2015 – M24
Actual submission date:	28 November 2015
Responsible Organization:	POLITO
Editor:	Cataldo Basile
Dissemination level:	Public
Revision:	1.0

Abstract:
This document presents the updates to the ASPIRE Security Model (ASM) and ASPIRE Knowledge Base (AKB), the advances in the Security Evaluation models and metrics, and the results of the empirical studies conducted with students of the academic project partners and with the tiger teams of the industrial partners.

Keywords:
ASPIRE Knowledge Base, security evaluation, empirical studies

Editor
Cataldo Basile (POLITO)

Contributors (ordered according to beneficiary numbers)
Bjorn De Sutter (UGent)
Cataldo Basile, Daniele Canavese, Leonardo Regano, Marco Torchiano (POLITO)
Mariano Ceccato, Paolo Tonella (FBK)
Paolo Falcarin, Elena Gómez-Martínez, Gaofeng Zhang (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

Disclaimer The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 609734. The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Executive Summary

This deliverable presents the achievements of the second year in WP4. Achievements are divided in three parts: the first part presents the updates to the ASPIRE Security Model and Knowledge Base, the second presents the updates on the metrics framework and to the security evaluation models, finally, the third one presents the results and planning of academic and industrial user studies. The delivery of D4.03 allows the ASPIRE project to achieve MS12.

Following the principles in D4.01, the ASPIRE Knowledge Base has been designed to include two types of information: a priori and execution-specific knowledge. A priori information includes concepts that exist regardless of ASPIRE project and regardless of the application to protect, like attacks, protection, assets, applications, tools, attack, metrics. Moreover, it includes concepts that exist regardless of the application to protect but they depend on the ASPIRE project, like the ASPIRE tool chain, protections developed within the ASPIRE project, ADSS preferences, annotations, etc. Execution-specific knowledge instantiates a priori concepts based on the information about the application to protect, like functions, variables, the actual assets and their links to the application code, attacks paths possible against the actual assets, protections that can be actually used, etc.

The ASPIRE Knowledge Base also supports reasoning about the concepts in the AKB. However, compared to planning in D4.01, there is major difference. The use of ontologies has been limited to statically represent the concepts in the ASPIRE Knowledge Base and a few DL classification reasonings. In the meanwhile, the Enrichment Framework, developed in WP5 and reported in D5.01, has allowed to externalize several reasoning thus achieving better performance and better development time.

The ASPIRE Security Model describes all the concepts used in the AKB. A priori concepts are in the main model. Then, several sub-models detail the following concepts: assets, application and their parts, protections and protection types, attacks, metrics, and protection requirements. Compared to the initial version (ASMv1.0) described in D4.01, the Y2 version (ASMv1.0) has very minor updates. The main model, and the asset, the metrics, the protection requirements sub-models have not been changed. There are four new classes and relations in the application sub-model. The purpose of these new entities is to describe code and data so that it is possible to reasoning about them when looking for the best solution (as needed by the ADSS). The attack sub-model introduces the attack target concept (and the needed associations) to better describe the concept attack goal and allow automatic attack paths discovery. The most significant updates are in the protection sub-model. First, this sub-model has been expanded by adding a new class to describe the protection profiles that are used by the ADSS, and new associations to depict protection dependencies. Then, we added a new class diagram that represents the annotations and the way they are organized and related to software protection. This class diagram describes the annotation design and representation made in WP5 in a way that is usable for logging annotation consumed by all the tools' activities in the ACTC.

The ASPIRE Security Evaluation combines the Petri Net models with the metrics framework to into the new Software Protection Assessment tool utilized to assess the protection strength based on the computed code complexity metrics.

The ASPIRE user studies are split between academic studies and industrial studies. Academic studies are controlled experiments conducted with students of the academic project partners. Industrial studies are case studies conducted with the tiger teams provided by the industrial partners. Academic studies aim at investigating the effectiveness of specific protection techniques, such as data obfuscation and code splitting. They analyse the factors that affect the effectiveness of protections and they consider different variants and configurations of the protections. Industrial studies evaluate the ASPIRE protections as a whole when applied to industrial strength software. They involve professional hackers recruited by the industrial partners.

Contents

List of Figures

List of Tables

1 Introduction

Section authors:
Cataldo Basile (POLITO)

Two of the most important objectives of the ASPIRE project are the design of protection techniques that are meaningful and effective to protect applications in the use cases, and the use of multiple lines of defence that strengthen each other to achieve a level of protection that is better than the sum of the individual protections. Moreover, the ASPIRE ambition is to use the ASPIRE Decision Support System (ADSS) to select the best combination of protections for preserving the assets in the application to protect.
To achieve these goals, several research and implementation questions need to be answered. First of all, we need to model assets, attacks, protections, dependencies among protections and among protections and attacks, etc. In short, we need to depict the landscape where ASPIRE plays its important role. Furthermore, we need to reason about these concepts, to automate the processes the ADSS must perform, thus the model must be designed to deal with logical inference, deduction, and abduction.
Then, we need to evaluate the effectiveness of the protections. Both metrics and simulation models join forces to have better estimation of the level of protection that can be achieved by deploying certain techniques against certain attacks on software assets. The most relevant improvement is to estimate the impact of the concurrent deployment of several techniques. However, these abstract evaluation models need to be proved against the abilities of real human beings, to check if (ex-post/ex-ante) estimates on the protection level are correct.
WP4 plays a crucial role in achieving these goals. Indeed, it creates the necessary infrastructure to achieve these goals and pushes the state of the art to solve the research issues.
Therefore, this deliverable presents the updates on:

- the ASPIRE Knowledge Base, the representation of the ASPIRE Security Model, designed to perform sophisticated reasoning (developed in T4.1 “Security Model and Evaluation Methodology”);
- the ASPIRE Security Model, which allows the formal representation of the ASPIRE concepts (developed in T4.1 “Security Model and Evaluation Methodology”);
- the metrics framework, designed to evaluate software protection strength through the use of software complexity and protection resilience metrics (developed in T4.2 “Complexity metrics”);
- the simulation models, designed to compare combination of protections, evaluate impact of protections and impact of attacks, et.c by means of Petri Nets and Montecarlo simulation models (developed in T4.2 “Complexity metrics”);
- empirical studies, which have conducted with academic and industrial parties to evaluate the effectiveness of the ASPIRE protections (developed in T4.3 “Experiments with academic subjects” and T4.4 “Experiments with industrial tiger teams”).

This deliverable delivers the achievements and progresses in WP4 at M24. The AKB and the Security Model can be considered stable, minor updates are expected at M30 in the deliverable D4.04 and possibly in the deliverable D4.06 at M36.
The metrics framework and the simulation models design is also very stable. However, these T4.2 activities will see important developments that will be documented mainly at M30 in the deliverable D4.04 with some minor update in the deliverable D4.06 at M36.
Results of all the experiments will be delivered in D4.04 and D4.06. Moreover, T4.5 activities about the public challenge will be delivered at M30 in the deliverable D4.05.

This deliverable is organized as follows. Part I presents the updates to the ASPIRE Knowledge Base. Section 2 concentrates on the updates on the design issues of the ASPIRE Knowledge Base. Section 3 reports the new version of the ASPIRE Security Model (ASMv1.1) and the changes compared to the preliminary version (ASMv1.0) documented in the deliverable D4.01.
Part II presents the updates to the ASPIRE Security Evaluation. Section 4 concentrates on the updates on the metrics framework. Section 5 reports the new Software Protection Assessment tool utilized to edit Petri Nets attack models and assess the protection strength based on the metrics computed by the metrics framework.
Finally, Part III reports the design and the results of the empirical studies conducted to evaluate the effectiveness of the ASPIRE protections. The ASPIRE empirical studies are divided into: (1) academic studies, controlled experiments executed by the academic partners, presented in Section 6 and 7; and, (2) industrial studies, case studies conducted by the industrial partners presented in Section 8.

Part I
The ASPIRE Knowledge Base

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

This part presents the ASPIRE Knowledge Base, the central aggregation of the ASPIRE knowledge, the place where all the ASPIRE information is described, represented, stored, and used to perform sophisticated reasoning.
After having introduced the design principles and compared them to the D4.01 preliminary design, this part describes the problems addressed and the solution for representing, manipulating, querying, and enriching the ASPIRE Knowledge base.
Moreover, this part introduces the new version of the ASPIRE Security Model, the M24 snapshot of the security model. This security model is built on top of the Preliminary ASPIRE Security Model presented in D4.01. For ease of notation, we named the Preliminary ASPIRE Security Model as ASMv1.0, and the current ASPIRE Security Model, the one presented here, ASMv1.1. Since it is much easier to present (and use) the entire ASPIRE Security Model in a single self-consistent document, this deliverable presents the entire model, not only the differences. However, each section related to the ASPIRE security model (i.e., Section 3 and all its subsections) starts with a description of the changes compared to ASMv1.0. According to the roadmap presented in D4.01, the content of this part, together with the deliverable D5.07, are the final delivery of the security model (D4.03 for the conceptual definition, D5.07 for the knowledge base enrichment).

2 The ASPIRE knowledge base

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

The ADSS relies on the knowledge in the AKB to perform its activities, which include the identification of the attacks that attackers can mount against application assets, and to identify the protections that can be enforced to mitigate the threats of those attacks. Moreover, by using this knowledge, the ADSS will use security evaluation features developed in the ASPIRE project (see Part II) to evaluate and measure the level of protection that can be provided by the available protections to mitigate the threats. The goal is to select the golden combination of protections that optimizes a set of user-defined criteria.
AKB has been designed to describe statically all the concepts needed by the ADSS and to represent additional data needed to reason about the knowledge available in the AKB. Therefore, AKB is provided as a conceptual model, the ASPIRE Security Model, which formally describes the concepts (i.e., the classes) required by the ADSS to work and their relations (i.e., the associations). More precisely, AKB contains classes to describe needed concepts listed, a complete object instantiation of those classes, which has been initially based on deliverable D1.02 (that informally describes this information) then extended and maintained by Consortium partners.
As anticipated in the deliverable D4.01, AKB represents a priori knowledge and application-specific knowledge. A priori knowledge comprises information that is independent from the program to protect. Execution-specific knowledge contains all the user 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.
Thus AKB includes (but it is not limited to):

- assets, threats, types of security properties to protect in applications;

- attacks, attack categories, attack tools, expertise/skill/resources needed to perform attacks, and consequences of attacks;
- software protections, their relations, incompatibilities, and synergies (i.e., the possibility to use them in combination to improve the overall protection);
- metrics to evaluate the probability of successful attacks, delays etc.;

The a priori knowledge formally represents what can be described, i.e., the classes and the associations. When these classes and associations are instantiated, we have execution-specific information.
However, a priori knowledge is not only formed of classes and associations. It also includes class instances that serve to describe information that is valid regardless of the application to protect. For instance, it includes known attacks, attack tools, and the relations between attacks and attack tools to mount them. It also comprises asset categories, security properties that users may require on assets, relations among attacks, assets and security properties affected by attacks and strengthened by protections.
Additionally, a priori knowledge allows the description of the following data:

- 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.;
- platform and other software-related data, OSes, processors and their instructions sets, emulators, interpreters;
- compilers, linkers and their options.

The initial purpose of describing abstract representations of the target applications in the AKB was to share the results of analysis tools among the ADSS and all the ACTC components. As agreed by Consortium partners, abstract representations are used by the ACTC components but not shared globally or made available to the ADSS. Therefore, this part of the AKB (that has not been extensively instantiated to execution specific knowledge) is still in an embryonal stage and will need further refinement in the unlikely possibility that these representations will be shared among components and the ADSS. However, there is an exception. The ADSS uses call graphs to determine relations among functions thus, relations among assets. This information is not shared with other tools, one Enrichment Module executes a static analysis tool to build the call graph and uses this information to identify attack paths against the assets.
The part of the a priori model to describe platform-related information, compilers and linkers seems expressive enough to cope with the ASPIRE needs. However, it has not been extensively used as we are concentrating on the ASPIRE use cases, where platform, OS, compilers and linkers have been selected by the Consortium, thus fixed.
A priori knowledge includes other concepts that only exist in the ASPIRE scope, for instance:

- 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 the 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.
- ADSS-related information, like ADSS-specific preferences, user constraints to consider during the selection of the best protection, i.e., limitations on measurable features that are altered during the application of protections (like code size, performance overhead, use of specific protections), pruning strategies (i.e., how to reduce the size of the problem space to be computationally feasible), etc.

Execution-specific knowledge complements the generic a priori knowledge to permit the ADSS to identify what to protect, decide which is the best protection to apply on the target application, drive the tool chain components when actually implementing the protections.
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 "component", (like functions, procedures, stubs, variables);
- the assets, corresponding to data or code regions, as provided by the user, and the corresponding threats;
- user ADSS characterization, that is the ASPIRE tool chain components available at the user installation ;
- 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.
- protection profiles, which summarize how the tools in the tool chain can be applied to protect the assets and the consequences of the application (performance degradation, network delays, bandwidth consumption, estimated security level).

The AKB is able to represent the logging information output by the ACTC when applying protections. This log includes the invoked protection tools, the actually used parameters, and the annotations consumed[1]. This ACTC output is currently processed by the security evaluation tools (to instantiate the Petri nets to simulate). It could also be used to fine-tune the configuration of protections applied later in the tool chain or as a feedback loop, to re-execute the ACTC. Moreover, this information 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 D1.03 v2.0).

Use of Ontologies for the AKB

The AKB has not been designed to be a static knowledge base that simply combines the a priori knowledge and the execution-specific knowledge. Instead, it is a dynamic knowledge base that is continuously improved as more knowledge becomes available during the operation of the ADSS. A priori knowledge has been modelled so that when execution-specific information is inserted, additional information can be deduced from it by means of ad hoc reasonings. Therefore an ADSS-driven enrichment process has been added on top of the conceptual model. The purpose is to minimize the manual input required from the user. Indeed, enrichment helps in completing the execution-specific knowledge.
Execution-specific knowledge enrichments in fact plays a major role. Starting from general information on attacks, protections, and relations to assets, AKB will infer protections and combination of protections that will work on the target application, given the assets and security properties requested by the user.
After the analysis at the beginning of this project, we adopted Description Logic (DL) ontologies to both represent the AKB and the enrichment processes needed by the ADSS to work (see D4.01 for further details on this analysis). Furthermore, we created an API to access the AKB ontology with basic Create Read Update Delete (CRUD) operations and support for complex queries. Both the AKB and the API have been made them available to the Consortium. To cope with the known

[1] Last version of annotation format is in the continuously updated WD5.02, it will be delivered in D5.11.

limitations of DL ontologies, we have designed and reported in D5.01 a modular enrichment architecture. Within this framework, reasonings that are not well performed with ontological method (like backward reasoning) or not possible because of the limitations of DL ontologies (like class level reasoning), are externalized as ad hoc enrichment modules. Enrichment modules only have to implement a common API and output the results of their inferences into the AKB.
Currently, we are using DL ontologies to represent the static information of the AKB and to perform simple reasoning (mainly classifications). The most powerful reasoning and enrichment have been all outsourced as Enrichment Modules. Enrichment Modules that are needed to determine the suitable combinations of protections are described in the deliverable D5.07. Final release of the ADSS Enrichment Modules will be delivered in M36 in the deliverable D5.10. This approach has shown several advantages: we achieve the best performance as we use the most suitable tools for each reasoning, we need less time to develop new reasonings as we use the most appropriate method. Moreover, we maximally exploit the ontology-based repository, API, and queries already developed and used by all partners to access the AKB.
As anticipated before, we have discovered that almost all the reasonings were better performed as Enrichment modules. Outside the ontology, where we can relax the DL and Open World assumptions, we were able to infer more information, better customize the reasoning, and in less (design and implementation) time. The most relevant case is the attack path discovery, based on backward reasoning, where we have built a very complex fact base based on Prolog.
Moreover, even if we have followed all the best practice, applied optimization tricks, and tailored our reasonings on the selected ontology reasoners (we used Hermit[2] that uses hyper tableaux algorithms and Pellet[3] that uses tableaux algorithms), inferences were better performed within the Enrichment Modules. Currently, only a few classifications and forward reasoning remain in the ontology, the ones that relate attack steps, attack tools, types of protections, and protections. Our scalability analysis showed us that performance were not acceptable for supporting protection decisions of real applications with several assets. To cope with this performance issue, we implemented a simplified 'adaptive merging approach', that dynamically included only the needed axioms from the required sub-ontologies, in order to improve performance of the reasoning process techniques by reducing the ontology "size". Even in this case, performance were not satisfactory. A sophisticated adaptive merging approach (to minimize the ontology size) would have required too much effort to be correctly implemented. Since ontologies were already fallen in disgrace, it has not been tried. In short, ontology reasoning was not as promising as we were expecting, thus we invested more effort in defining and optimizing reasonings in external Enrichment Modules. However, ontologies still have an important role in the AKB to store static information. Indeed, 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 / UML class diagrams. Additionally, the performance of ontologies for querying data were satisfactory for our purposes.
A further clarification is needed to explain how the AKB supports the Enrichment Modules. Enrichment Modules must not maintain static information. That is, all the information needed to perform the reasoning (in other words, the facts) must be retrieved from the AKB. Only, inference rules and model building rules are in the Enrichment Module. For this reason, additional information used by Enrichment Modules that violates the DL assumptions or it is not usable by the ontology reasoners, has been encoded into the ontology as additional data properties (of String type), associated to ontology individuals. When retrieving ontology individuals, Enrichment Modules also access this additional information and also have the responsibility to know the existence of these fields and correctly interpret them. As an example, additional facts concerning the attack steps (individuals) used by the Prolog-based Enrichment Modules are conveyed by means of the `prologFacts` data property. In general, an unlimited number of data properties can be added to the ontology to support analogous cases.

[2] `http://www.hermit-reasoner.com/`
[3] `http://clarkparsia.com/pellet/`

3 The ASPIRE security model

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

3.1 The ASPIRE security model v1.1: main model

Changelog: No changes compared to ASMv1.0.

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 1 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 3.2.2. 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 3.2.1, where the `Application` sub-model is presented.
`Assets` instances 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.
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

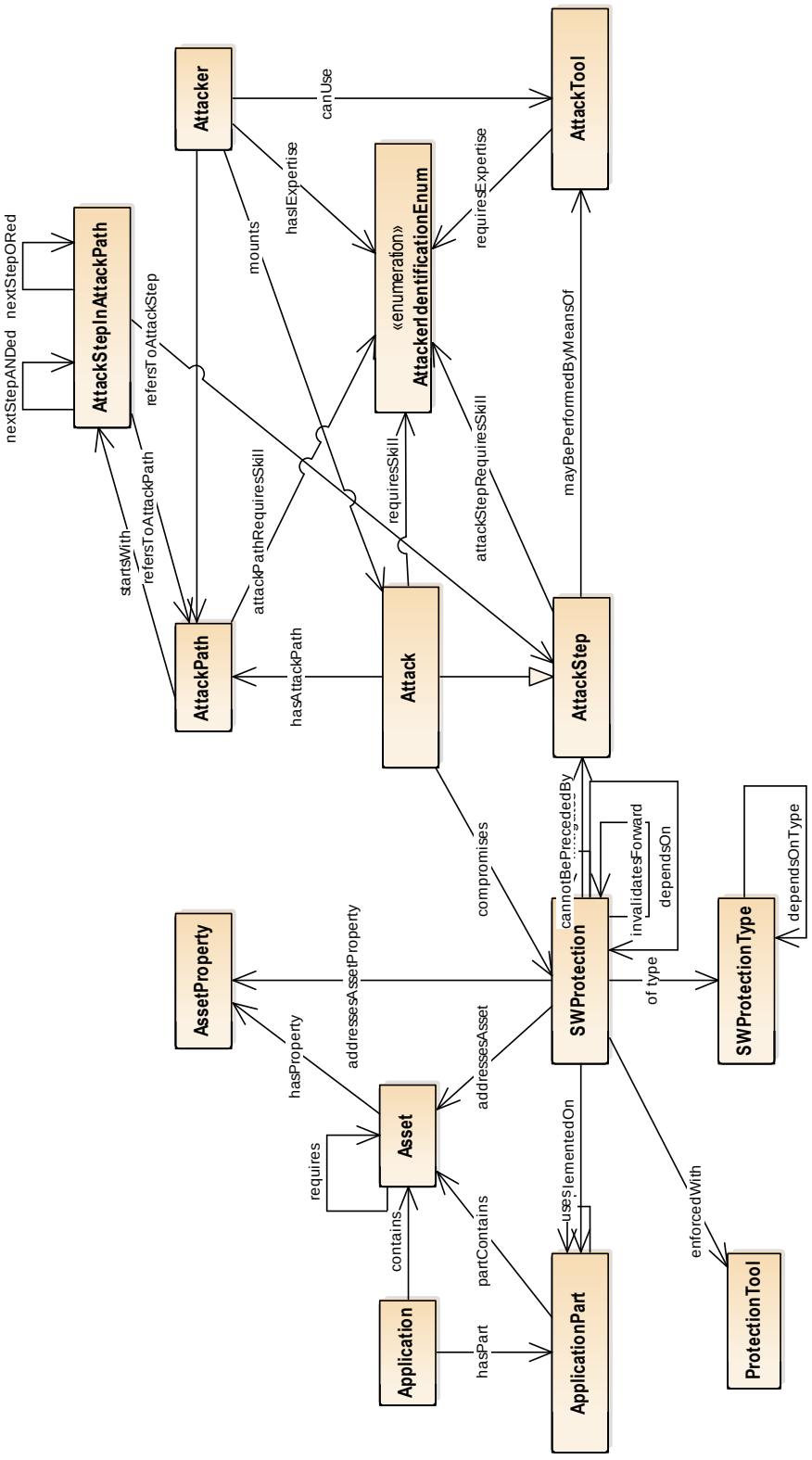


Figure 1: The main model.

(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, that is, these associations allow us to model graphs of attack steps (and Petri nets), not only sequences (see Section 3.2.4).

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 do not 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 values of the AttackerIdentificationEnum enumeration 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 preconditions to apply the later one. Software protections are categorized in types by associating them to SWProtectionType instances via the ofType association. A software protection is associated to exactly one SWProtectionType instance.

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 3.2.3). 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 3.2.6) 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 3.2.6).

3.2 Model Extensions: the Sub-Models

Changelog: The asset, the metrics, the protection requirements sub-models have not been changed. The application, attack, and protection sub-models present new classes and/or associations. All the changes are described at the beginning of each sub-model presentation. Moreover, in all the figures, new associations are depicted with a thicker line and new classes have a thicker border.

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.

3.2.1 Application sub-model

Changelog: With respect to the ASM v1.0, the newer model shows very few changes, essentially only in the `ApplicationParts` package. In detail, four new entities were added: the classes `Code` and `Data`, added for easing the reasoning in the AKB, and the associations `contains` and `uses` used to relate the application parts together.

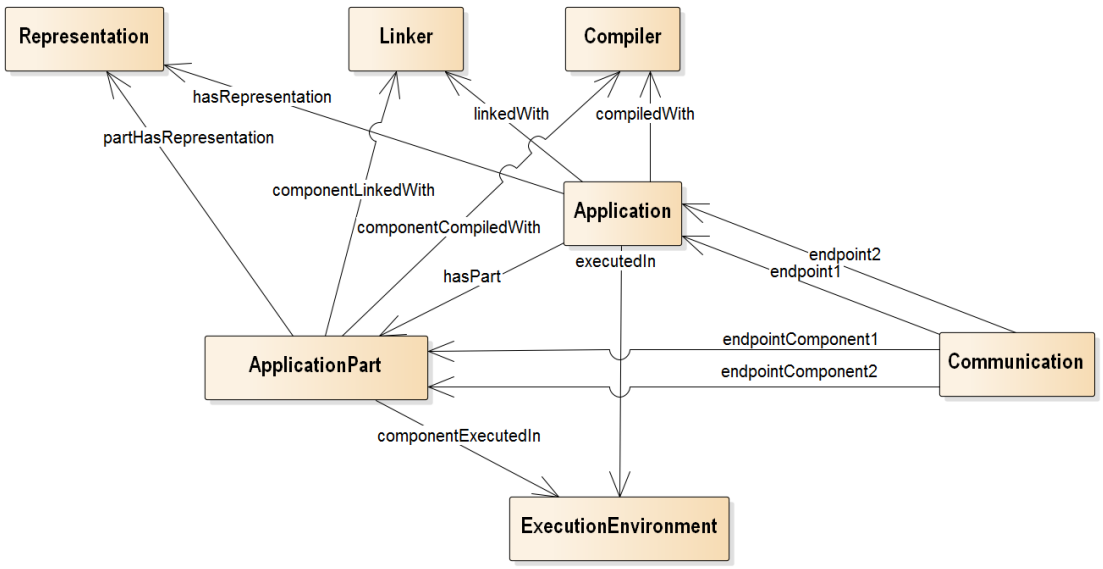


Figure 2: The applications sub-model.

The sub-model shown in Figure 2 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 with 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 2 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;

- `ExecutionEnvironment` instances where the application can be run via the `executedIn` association;
- `Representation` instances, that abstractly describe these application parts via the one-to-many associationhasRepresentation association;
- `Compiler` and `Linker` instances used to compile it, via the associations `compiledWith` and `linkedWith`, which are valid if the single application components are not compiled independently;
- `Communication` instances of which the `Application` is an endpoint, will be identified by navigating the `endpoint1` and `endpoint2` associations in opposite directions.

It is worth noting that the `ApplicationPart` instances are also linked with the `Application` instances in the `ApplicationParts` package. The `ApplicationParts` package presented in Figure 3 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:

- `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);
- `Library` class, used to describe static or dynamic libraries, represented as `StaticLibrary` and `DynamicLibrary` class instances. `DynamicLibrary` instances are connected to instances of the `LibraryModule` class by means of the `containsLibraryModule` association;
- `Code` class instances contain all the kind of executable code (functions, methods, snippets, ...) than can be found inside an application. Note that a `Code` instance can contain other application parts (e.g. a function contains a PIN or a code fragment). This is modeled by the `contains` association. Furthermore, this class has several specialized sub-classes:
  - `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 (related by means of the `includesAlgorithm` and `includesProcedure` associations);
  - `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;
  - `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.
- `Data` class models all the kind of static/dynamic data that an application can contains. Currently it has only one sub-class, `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

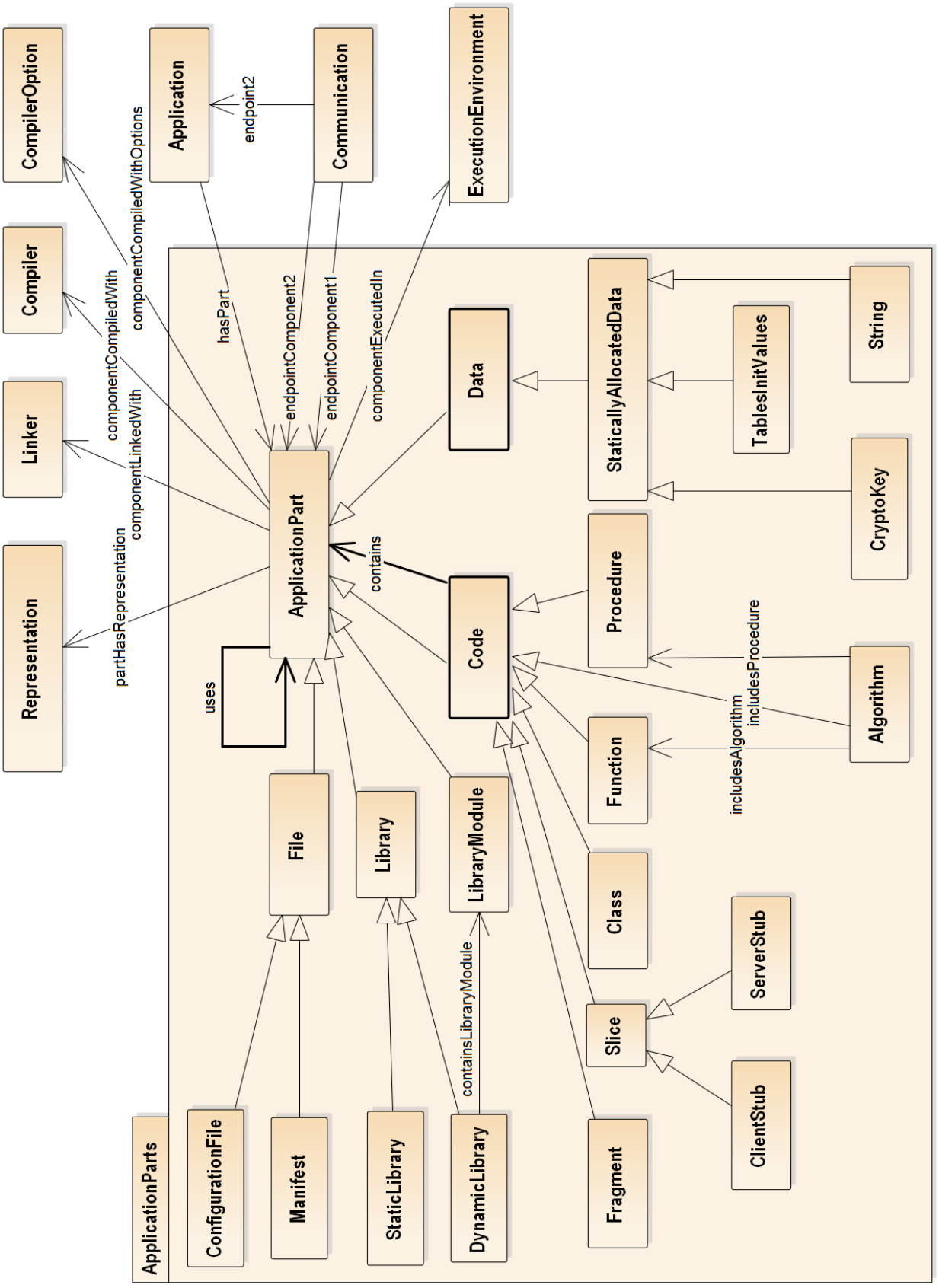


Figure 3: The `ApplicationParts` package.

are strings, which attackers look after during certain attack steps, modelled with an ad hoc sub-class `String`.

The `ApplicationPart` instances are connected to:

- themselves via the `uses` association. This relationship is used to model both data exchanges and flow-execution jumps between functions, snippets, ...
- `Application` instances are part of (via the one to main `hasPart` association);
- one or more `ExecutionEnvironment` instances that describe where applications can be run via the `componentExecutedWith` association (as some application is made of several independently executing component, like clients and servers);
- `Communication` instances they are endpoint of (via the one-to-many associations named `endpoint1` and `endpoint2`);
- `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 word "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.

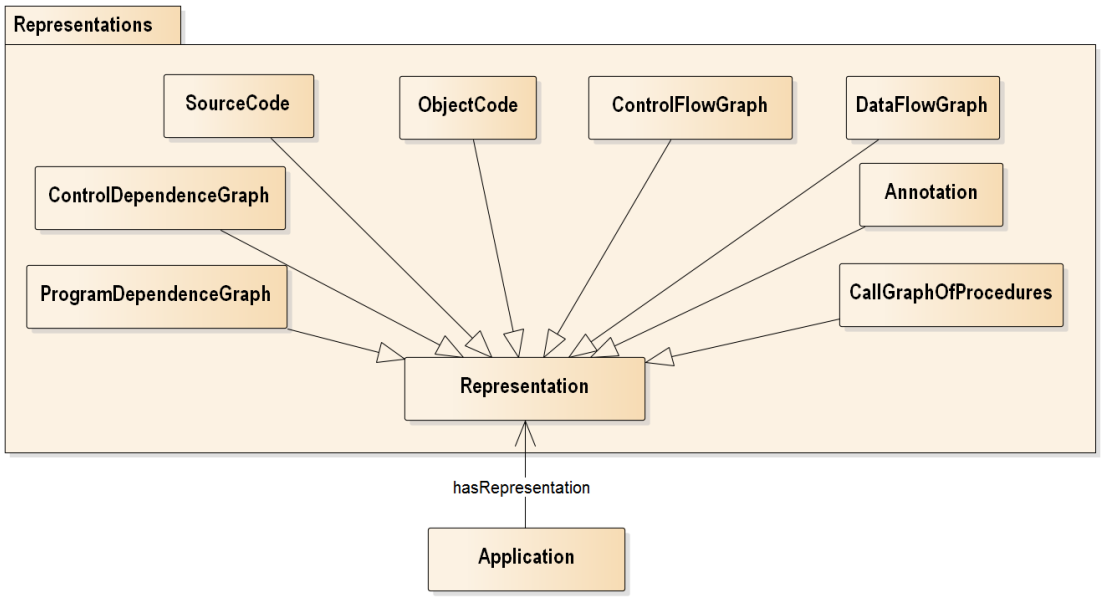


Figure 4: The `Representations` package.

The `Representations` package (see Figure 4) includes all the abstract representations that can be used by the ADSS or tool chain components. It includes the abstract `Representation` class which will be sub-classed 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.

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).

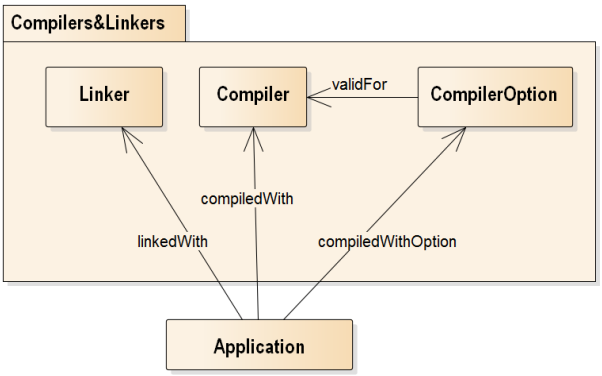


Figure 5: The `Compilers&Linkers` package.

For the `Compilers&Linkers` package depicted in Figure 5, 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 also 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 in order to derive some general information of the object produced and may serve to the decision process. The ASPIRE Consortium is still debating the need to include this information in this package.

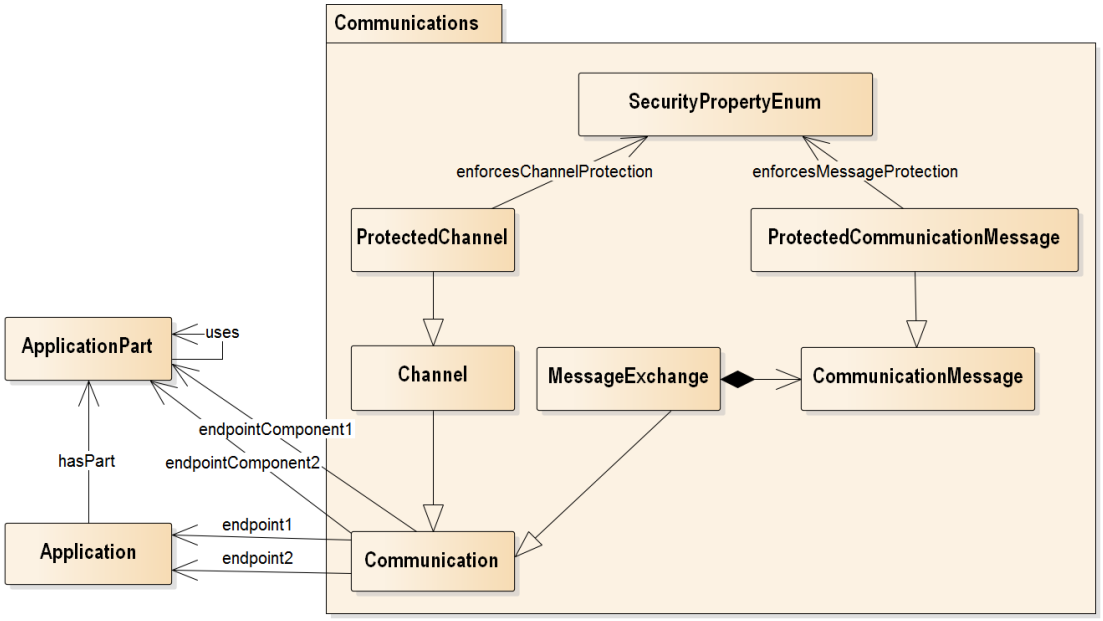


Figure 6: The Communications package.

Figure 6 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 sub-classed in the Channel and the MessageExchange classes. The Channel class has been further sub-classed in the ProtectedChannel class to describe protected channels. Instances of the ProtectedChannel class are bounded to the security properties they guarantee by means of an ad hoc association: enforcesChannelProtection. Security properties are represented by means of instances of the SecurityPropertyEnum class, an enumeration listing all the possible security properties that can be enforced on communications (e.g. integrity and confidentiality). 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 (for instance, message authentication, integrity, and confidentiality), we use an ad hoc class : ProtectedCommunicationMessage. Its instances are also connected to instances of the SecurityPropertyEnum c;ass, via the association named enforcesMessageProtection. 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 instances of the ApplicationPart class, 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.

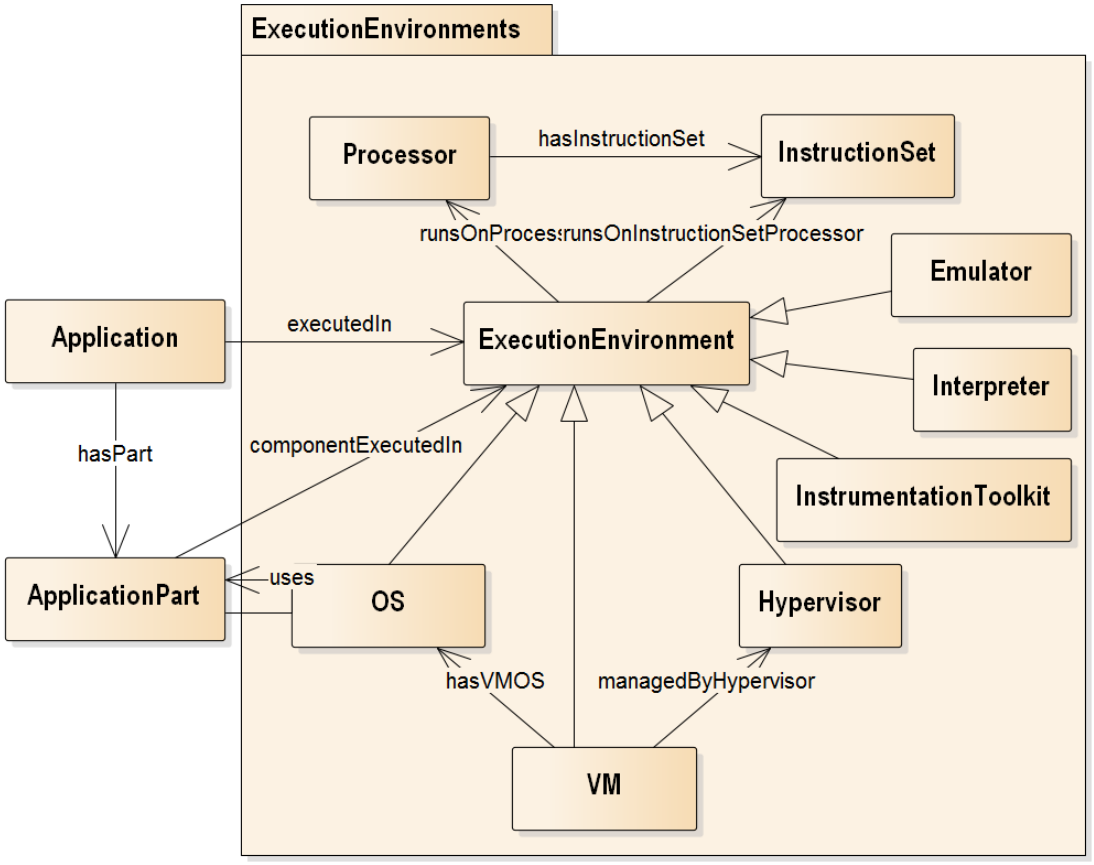


Figure 7: The `ExecutionEnvironments` package.

Finally, the `ExecutionEnvironments` package, depicted in Figure 7, refines the abstract class `ExecutionEnvironment` by sub-classing 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.

3.2.2 Assets sub-model

Changelog: No changes compared to ASMv1.0.

Figure 8 reports the categorization of assets presented in D1.02. The different asset categories have been modelled with the sub-classing paradigm. 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 sub-classed in `SecurityLibrary`, `CustomAlgorithm`, and `PrivateProtocol`.

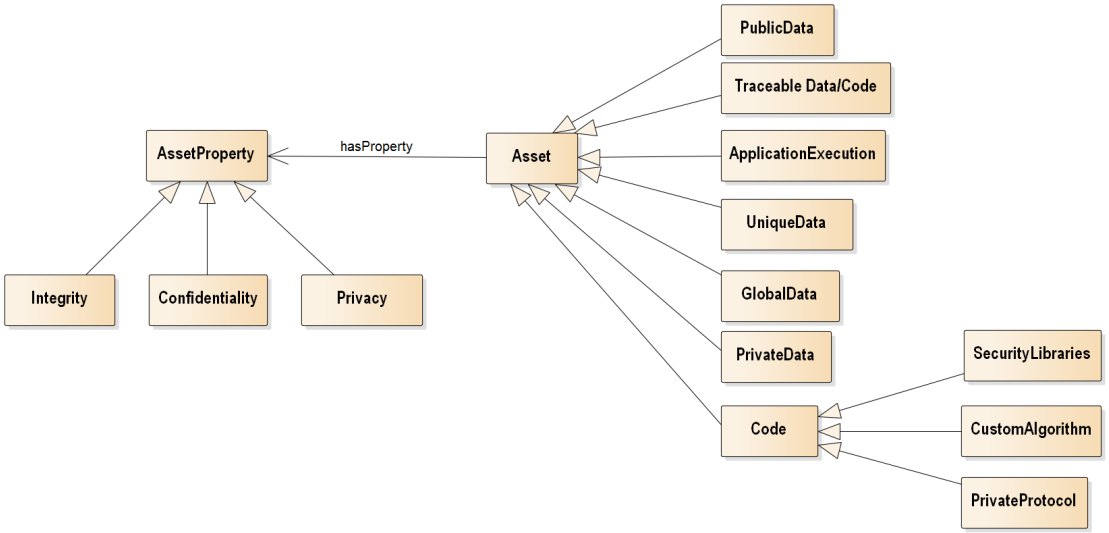


Figure 8: The asset sub-model.

3.2.3 SW Protection sub-model

Changelog: The SW protection sub-model was expanded by adding a new class (`SWProtectionProfile`) and two new associations (`cannotBePrecededBy` and `hasProfile`). Moreover, we added an entire sub-model part to describe annotations.

Figure 9 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 sub-classed in `Source2SourceDataObfuscation` and `WBC` (white-box crypto);
- `AlgorithmHiding` has been sub-classed in:
  - `SourceLevelAlgorithmHiding`, further sub-classed in `PatternRemoval`;
  - `ClientSideVM`;
  - `BinaryCodeObfuscation`, further sub-classed in `CodeFlattening`, `BranchFunctions` and `OpaquePredicates`;
- `AntiTampering`, has been sub-classed in `AntiCodeInjection`, `AntiLibraryCallback`, `AntiDebug` and `CodeGuards`;
- `Renewability` has been sub-classed in `RenewabilityInSpace` and `RenewabilityInTime`
- `ReactiveTechnique` has been sub-classed in `TimeBombs`.

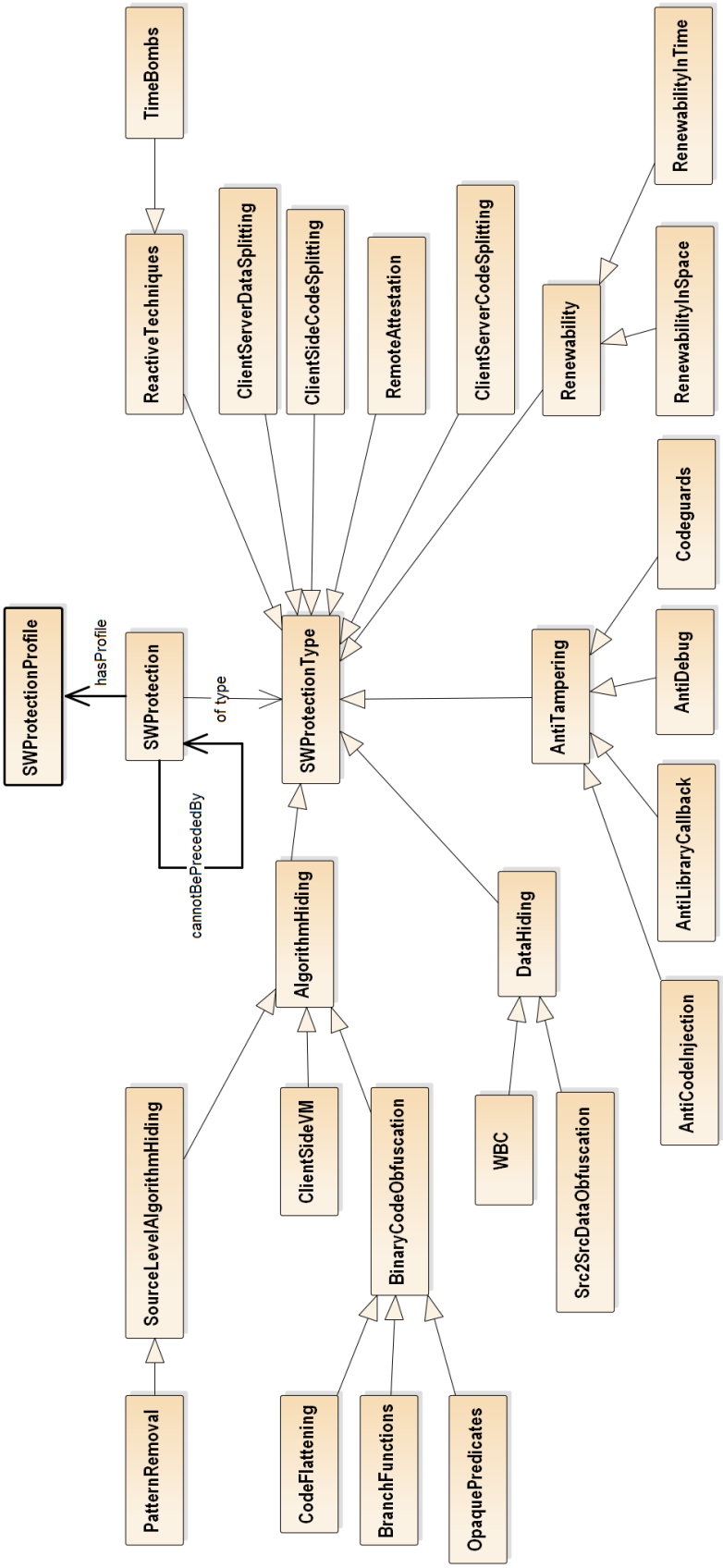


Figure 9: The SW protections sub-model.

Integrity protection techniques, like remote attestation and anti-tampering techniques, are also categorized as either static and dynamic. However, instead of using the sub-classing paradigm we plan to add a Boolean attribute (`dynamic`) in those classes.
A protection technique exposes one or more protection profiles (see D5.01), implemented as instances of the `SWProtectionProfile` class. The `hasProfile` annotation is used to link the profiles to their related `SWProtection` objects.
Some protection techniques cannot be applied on the same asset in any order, so that a (partial) ordering is needed. The ASM models this by making use of the `cannotBePrecededBy` association, used to express the fact that a protection technique cannot used after another one.

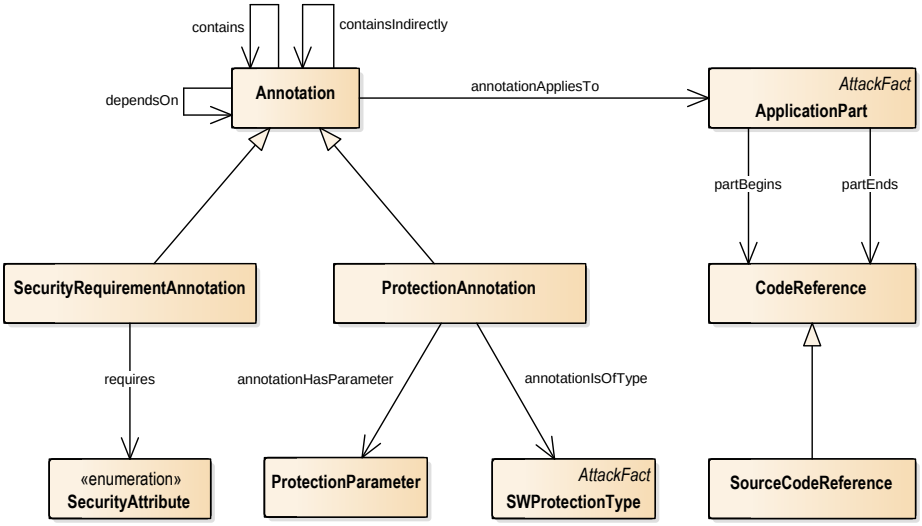


Figure 10: The annotation part of the SW protections sub-model.

The main class of this part of the SW protections sub-model is the `Annotation` class (see Figure 10). All instances of the `Annotation` class contain the attribute `annotationID` to univocally identify them (within the application to protect, of course).
Annotations apply to specific part of the application to protect. These parts are represented with `ApplicationPart` instances available in the `Application`. To univocally identify the application parts, we introduced the `CodeReference` class. Each `ApplicationPart` instances is associated to two `CodeReference` instances: one indicates where the part begins and the other one where the application part ends. This scenario is modelled through the `partBegins` and `partEnds` associations. Currently, we only need to refer to source code, therefore, file names and the line numbers where the part starts and ends are enough to describe the application part. Therefore, we subclassed the `CodeReference` into the `SourceCodeReference` class, which contains the `filename` and `line_number` attributes.
Moreover, there are several associations to express dependencies among `Annotation` instances. These dependencies will be exploited by the ADSS or protection tools able to process them. The `contains` association is used to describe nested annotations[4] The `containsIndirectly` association to represent annotations that are reached when the code within an annotation is executed. For instance, the fact that an `Annotation` instance a_2 in a function called from within the application part associated to the another `Annotation` instance a_1 is represented by associating a_1 `containsIndirectly` a_2. Finally, to represent dependencies among annotations, e.g., to relate annotations that are part of the same technique, like parts protected with the same guards, or different parts that need to be protected analogously, the `dependsOn` association has been introduced. The exact semantics of this association depends on the technique that will actually consume/use it.

[4]It is worth noting that the ASPIRE consortium has decided that valid annotations can be either nested or disjoint. No cases of partly overlapping annotations is possible. Moreover, they have to start and end in the same file.

Annotations are divided in security requirement annotations and protection specific annotations (see deliverable D5.01). Therefore, we introduced two subclasses of the `Annotation` class, the `SecurityRequirementAnnotation` and `ProtectionAnnotation` classes.
`SecurityRequirementAnnotation` instances reflect the security requirements defined in the deliverable D1.04, thus, `SecurityRequirementAnnotation` instances point to instances of the `SecurityAttribute` enumeration class through the `requires` association. These enumeration includes the following values: `confidentiality`, `integrity`, `privacy`, `nonRepudiation`, `executionCorrectness`.
`ProtectionAnnotation` instances refer to the `SWProtectionType` instances they ask to enforce on that application part (via the `annotationIsOfType` association). Moreover, `ProtectionAnnotation` instances link all the protection specific parameters that need to be passed to the protection technique to be enforced according to the software engineer in charge for protecting the application (or the ADSS) by means of the `annotationHasParameter` association. These parameters are characterized as name, value pairs. Therefore, `ProtectionAnnotation` instances are associated to instances of the `ProtectionParameter` class, which contains two attributes, `name` and `value`.

3.2.4 Attacks sub-model

Changelog: The ASM v1.1 attack sub-model contains only two new entities: the `AttackTarget` class and the `hasTarget` association, used to related the attack target instances to an attack.

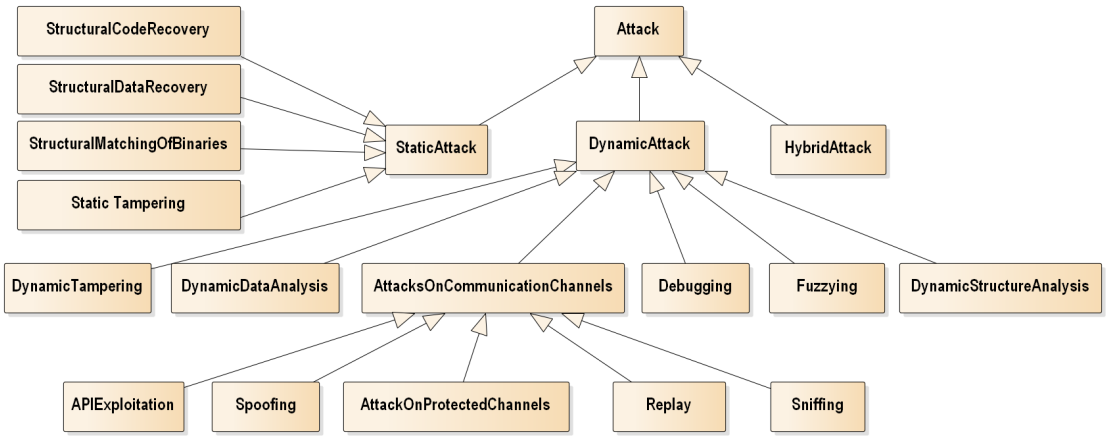


Figure 11: The attacks sub-model.

Figure 11 presents the attack sub-model. The categorization of the attacks comes directly from D1.02, that distinguished static, dynamic, and hybrid attacks. To that extent, we introduced the top-level sub-classes `StaticAttack`, `DynamicAttack`, `HybridAttack`, `PassiveAttack`, and `ActiveAttack`.
Static attacks are further sub-classed in `StructuralCodeRecovery`, `StructuralDataRecovery`, `StructuralMatchingOfBinaries`, and `StaticTampering`. Dynamic attacks are further sub-classed in `DynamicTampering`, `DynamicDataAnalysis`, `Debugging`, `AttacksOnCommunicationChannels`, `DynamicStructuralAnalysis` and `Fuzzying`. `AttacksOnCommunicationChannels` has been further sub-classed in `APIExploitation`, `Spoofing`, `Sniffing`, `Replay` and `AttacksOnProtec-tedChannels`.
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 12 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

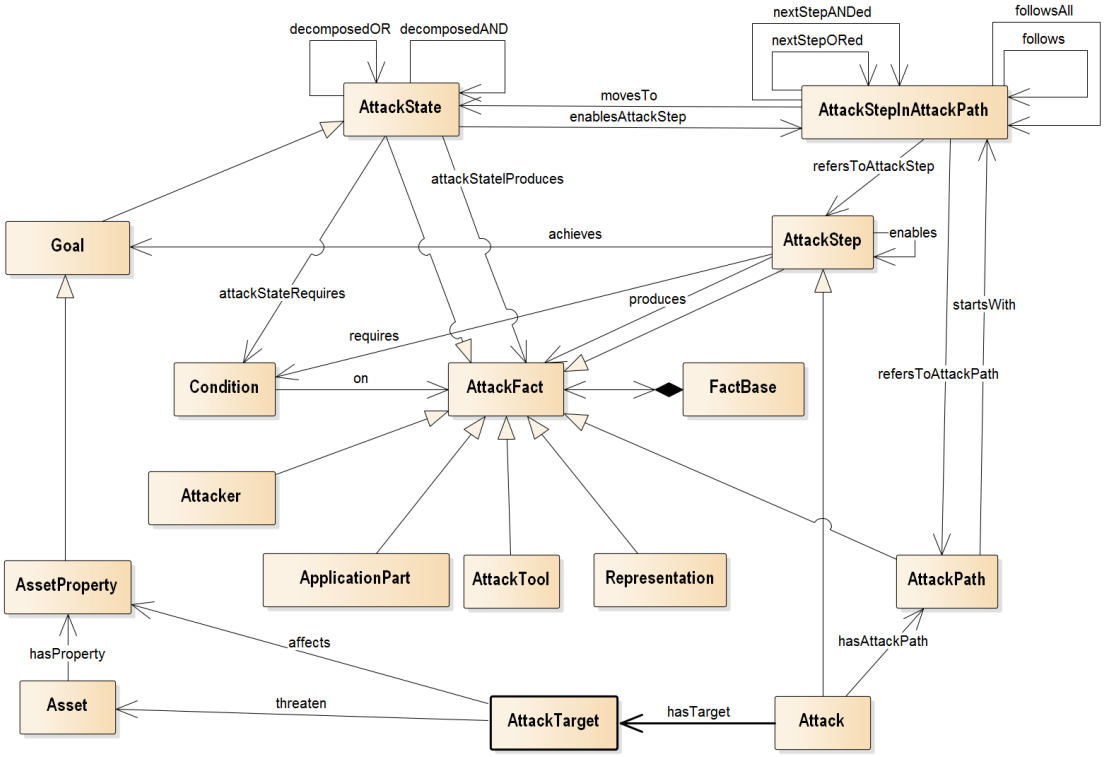


Figure 12: The attacks, goals and Petri nets.

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`, which refers to the `AttackStep` via the `refersToAttackStep`. 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.
`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 sub-model: 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.
An Attack class instance is related to one or more asset since its 'job' is to disrupt some assets' security property. This is modeled by the AttackTarget class and the hasTarget one-to-many relationship.

3.2.5 Metrics sub-model

Changelog: No changes compared to ASMv1.0.

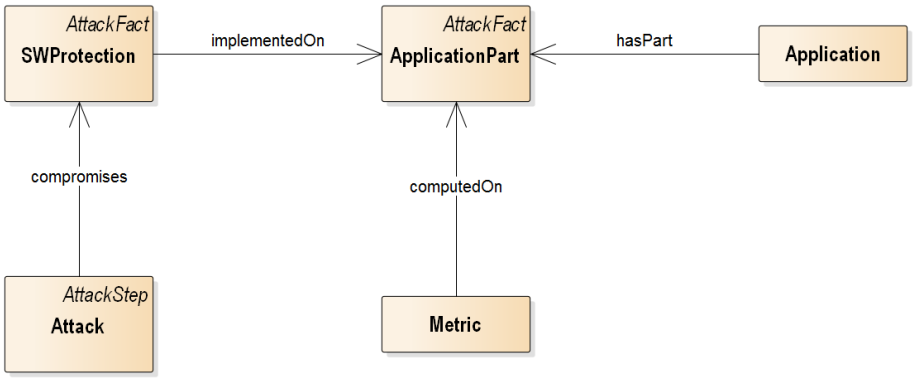


Figure 13: The metrics sub-model.

The metrics sub-model, depicted in Figure 13, considers the relations among metrics, represented by the Metric, application parts, and attacks. 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, the metrics will be used to quantify the impact of a protection, applied on an application part, against a specific attack. For this purpose, we make use of the `evaluatesImpactAgainst` association 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.

3.2.6 Protection requirements sub-model

Changelog: No changes compared to ASMv1.0.

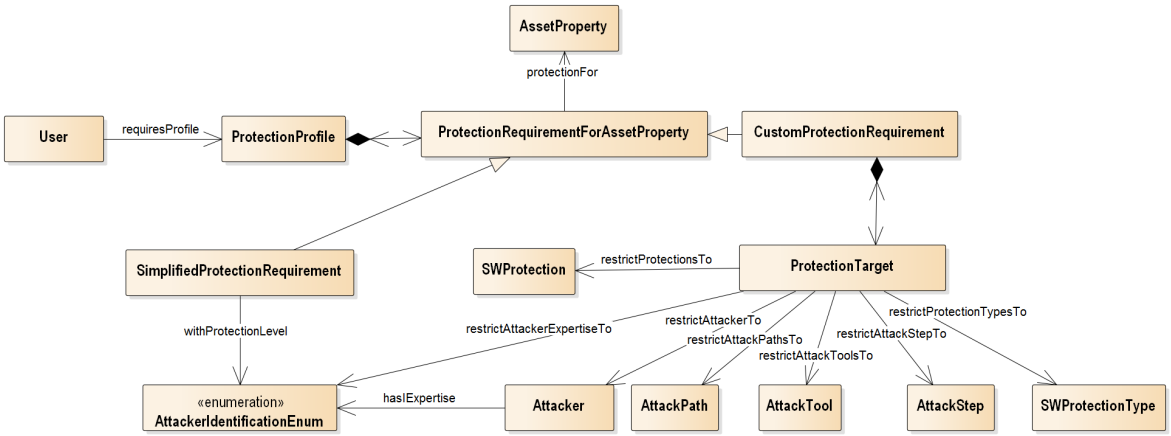


Figure 14: The protection requirements sub-model.

One of the ASPIRE project 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. Figure 14 shows the classes, and their associations, used to model this situation.

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 sub-classed `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. The attackers to protect against can be defined as an `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 `AttackPath` 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. A `CustomProtectionTarget` instance is associated to the `SoftwareProtection` or `SoftwareProtectionTypes` instance the user wants to consider during the selection of protections (via the `restrictProtectionsTo` and `restrictProtectionTypesTo` associations).

Part II
Security Evaluation

This part introduces the new framework for computing software complexity metrics and the Software Protection Assessment tool created to edit the attacks models, used with the metrics to evaluate the strength of the types of software protections integrated in the ASPIRE Toolchain.

4 Tool Support for Computing Software Complexity Metrics

Section authors:
Bjorn De Sutter (UGent)

At the end of the first year of the project, deliverable D4.02 Part II introduced a whole new framework for computing software *complexity metrics* and so-called software *resilience metrics* that would be useful to evaluate the strength (potency, resilience, and stealth) of the types of software protections developed and integrated in the ASPIRE project against a wide range of attacks.
In this section, we report on the follow-up work that has since been performed with regards to those two types of metrics.
Furthermore we report the progress that was made in year 2 regarding an alternative method of evaluating protection strength. That alternative method, as described in the DoW, comprises the use of real-life attacker tools such as IDA Pro and Bindiff, using automation scripts to mimick the attackers' use of those tools and automated techniques to quantify the value of the tools for the attacker. That quantification is performed by comparing the tools' output to the ground truth (known to the user of the protection tool), and by quantifying the outcome.

4.1 Automated support for tool-based metrics

In this line of research, we can report progress on two fronts.

4.1.1 Diffing tools

First, we migrated and extended UGent's existing background for semi-automated diffing of x86 binaries on Windows hosts to fully automated diffing of ARM binaries on Linux hosts.
This approach consists of a number of scripts that invoke Diablo (the link-time rewriter in which most of the binary-level processing steps of the ACTC are implemented), IDA Pro and BinDiff. Based on a binary or library i Diablo produces two different, protected binary versions o_1 and o_2, along with two mapping files m_1 and m_2 that describe where in o_1 and o_2 all of the assets of the input binary i ended up.
IDA Pro is then invoked to disassemble the two binaries o_1 and o_2 like an attacker would do in a collusion attack on two program versions (e.g., one original binary and one renewed binary), and BinDiff is invoked to find the matching fragments. Using some custom scripts we developed to serialize the data that is normally visualized in a GUI, the invocation in BinDiff produces a mapping m_{bd} in a file.
Using the ground truth encoded in m_1 and m_2, automated scripts then determine two scores for BinDiff. First, the scripts compute how many code fragments BinDiff matched correctly between the versions o_1 and o_2.
Secondly, the scripts compute whether or not the diffing results are sufficiently accurate to match the assets in both binaries. This way, the scripts compute whether or not the differ helps an attacker in finding assets in o_2 if he would have already found them in o_1 during a previous attack. Furthermore, the scipts compute whether or not the diffing tools help an attacker in identifying the differences in the protections applied to some asset in o_1 and o_2. The scripts can take into account

multiple attacker heuristics and mimick the behavior of an attacker. For example, if the diffing tool would show the control flow graphs of two matched functions in the two binaries on screen, and highlight the differences in the two functions' graphs, the attacker would inadvertently also see the neighbouring code of the highlighted differences. So even if the differ does not highlight a meaningful difference between o_1 and o_2, the attacker's attention might already be drawn to it as a side effect.
This effect and several heuristics an attacker might deploy to focus his attack on the most relevant program fragments are incorporated in the scripts, and can be disabled/enabled easily to model different attackers with different skill sets.
The overall approach was already described in literature by UGent. In this project, UGent implemented the necessary support to integrate the approach and to automate it in the context of the ACTC and the project demonstration and validation on the ARM Android platform.

4.1.2 Disassemblers and control flow reconstruction

One of the most basic attacker tools are disassemblers like IDA Pro: they identify code bytes in a binary or library text segment, disassemble the code bytes into instructions, partition those into basic blocks, and then partition those in functions while reconstructing those functions' control flow graphs.
Thwarting the tools with respect to all of these steps is one of the main goals of binary code obfuscation. In the ASPIRE project, this obfuscation is delivered by means of binary code control flow obfuscations (Task T2.4 in WP2), by replacing native code by bytecode to be interpreted (Task 2.3 in WP2), and by means of code mobility (Task 3.1 in WP3).
To measure the effectiveness of a disassembler, the notion of a confusion factor was introduced in literature. See Section 5.1.3 in D4.02 in this regard. That metric models the fraction of the static code that can be disassembled correctly using state-of-the-art static, recursive-descent disassemblers (like the one from IDA Pro). On the ARM architecture, however, with its fixed word width, a thus defined confusion factor makes little sense, because all code is aligned.
Instead, we therefore aim to measure the more abstract goal of tools like IDA Pro to reconstruct control flow graphs correctly. To that extent, we developed the necessary scripts on top of IDA Pro and Diablo. When Diablo is invoked (as the final step in the ACTC) to protect an application, it outputs a ground truth mapping m_p of instructions to functions. In essence, this maps defines which instructions belong together in a function.
The scripts invoke IDA Pro to compute control flow graphs of all functions, and let IDA Pro output a list of functions e_p, indicating which instructions it grouped per function. The scripts then compare e_p to m_p to compute how many of the instructions were correctly assigned to the same function by IDA Pro. Furthermore, the scripts count the number of functions supposedly found in the binary code by IDA Pro.
In deliverable D2.06, Section 4.2, which was delivered at M18, the strength of Diablo's binary control flow obfuscations was already evaluated by means of these scripts and metrics.

4.2 Automated Tool Support for Complexity Metrics

Section 5.1 of deliverable D4.02 defines a number of static complexity metrics to be computed on graph representations of the programs, and a number of dynamic complexity metrics to be computed on program traces. For a number of them, we developed a concrete formulazation in the form or a reference implementation in Diablo and in external scripts, such that the metrics can now be computed on protected libraries and binaries generated by the ACTC.

4.2.1 Static metrics

As for the static metrics, Diablo now features support for three different metrics:

- Halstead's program size metric that combines the number of operations and the number of operands in a static program representation. This is the SPS metric as defined in deliverable D4.02 Section 5.1.1.
- Cyclomatic number, a metric for the complexity of control flow graph structures. This is the CC metric as defined in deliverable D4.02 Section 5.1.3.
- The number of indirect control flow branches in the program, (excl. standard switch statements). This is the CFIM metric as defined in deliverable D4.02 Section 5.1.3.

These metrics are now always computed for any program protected with the ACTC. Moreover, they are computed per code region corresponding to an ASPIRE source code annotation. This allows the ADSS to evaluate the strength of the tried protections per specific asset in the code.

4.2.2 Dynamic metrics

We also developed the necessary support in the Diablo rewriter for one of the most fundamental dynamic metrics: the dynamic program length or DPL as defined in Section 5.1.2 of deliverable D4.02.
To increase the flexibility with which these metrics can be computed, both by users of the ACTC and by automated tools like the ADSS, we developed two mechanisms to compute the metrics.
In the first mechanism, an unprotected binary is first profiled. To that extent, Diablo now includes the necessary profiling support for ARM Android & Linux binaries. It suffices to pass the binary or library through Diablo without deploying any protections to generate a version of the binary or library that can profile itself. To achieve this, Diablo instruments the program as follows:

1. Diablo inserts a static initializer function into the binary to reset the basic block counters;
2. Diablo inserts code to increment a basic block counter upon entry to each basic block in the program;
3. Diablo inserts a static finalizer function that ensures the collected basic block counters are written to a file on disk when the execution of the binary or library finishes.

In a second run of Diablo (i.e., of the ACTC invoking Diablo), Diablo will read the collected profile information. In this run, Diablo also deploys all requested protections, for which it repeatedly transforms the program's control flow graphs. While doing so, it keeps track of the basic block execution counts. So after the final protection is applied, Diablo can estimate rather precisely how many times each block will be executed in the protected binary (assuming, of course, that the used profile inputs were representative). On the basis of these execution count estimates, Diablo then outputs the dynamic metrics.
To make this first mechanism more useful in the context of ASPIRE, the necessary bookkeeping support was also developed to translate profile information from one version of a binary to other versions. In particular, a user can collect profile information on one version, and reuse that information in another version that is identical to the first version, but with additional objects linked into it. In the context of ASPIRE, this is useful, because the linking step in the ACTC not only links in all object files of the original application or library, but also a number of protection libraries such as the SoftVM, the ASPIRE Common Communication Logic, the remote attestation routinges, the Code Mobility Binder and Downloader, etc.
In the second mechanism, no profiling version of an unprotected binary is generated. Instead, Diablo applies the protections right away, after which it produces two binaries: a standard protected binary, and an instrumented protected binary. The latter one is nothing more than a version of the standard protected binary, augmented with the instrumentation code. In addition, Diablo produces some auxiliary files that store, per basic block, the relevant features to compute dynamic complexity metrics. When the protected, instrumented binary is executed, it will write out basic block counts to a file, just like a non-protected version would do.

A number of Python scripts developed at UGent then combine the stored basic block counts with the information in the auxiliary files to compute the dynamic metric.
The advantage of the second approach is that it requires only one run of Diablo: the run in which the protected binary is generated, together with the instrumented version thereof. The downside is that for every protected version for which one wants to collect the metrics, a new profiling run using the protected binary is needed.
In the first approach, the situation is exactly the opposite: only one profiling run is needed, on the unprotected binary. In some cases, though, the first approach might produce inaccurate results, because it is not always possible to perform accurate bookkeeping of the execution counts while the program is being transformed.

4.3 Automated Tool Support for Resilience Metrics

Section 5.2 of deliverable D4.02 introduced the notion of resilience metrics. Two of the proposed metrics, Intra-Execution Variability (Section 5.2.2 of D4.02) and Semantic Relevance (Section 5.2.3 of D4.02) are to be computed on the basis of full execution traces.
At the time of writing of this deliverable, we still lack full tracing capabilities. Those are planned to be developed in the first months of year 3 of the project.For the time being, we therefore use the existing profiling capabilities, as discussed in the previous section, as a first-order approximation. We do so in two ways.
First, we use the profile information consisting of basic block counts to evaluate which conditional branches in the program display intra-execution variability. Concretely, we use the basic block execution counts to evaluate which conditional branches are executed in both directions (branch taken and branch not taken) during the execution of the program. As discussed in Section 5.2.2 of deliverable D4.02, this is an important property in light of trace-based attacks that replace quasi-invariant instructions by simpler variants.
Secondly, we use the basic block execution counts as an indication of semantic relevance. The simplified rule or underlying assumption is that any block that is executed in the program contributes to the program's semantics, while any block not executed (such as code checking for normally not occurring out-of-memory scenarios) does not contribute to the program semantics.
While this is only a rough estimation of true semantic relevance, that can certainly not match the taint-based approach of Debray et al (see the updated D1.04 v2.0, Section 4.4.4.2), we still consider it useful. As already noted in D4.02 Section 5.1.2, a natural extension of the dynamic code size metrics (and of any dynamic metric for that matter) consists of coverage-based metrics. In such metrics, features of executed (i.e., covered) fragments are counted, but not weighed by execution counts. This is exactly what our first-order approximation of semantic relevance achieves: covered code is considered semantically relevant, non-covered code is considered semantically irrelevant.
For all of the mentioned static and dynamic metrics of Section 4.2, our tools therefore also report a coverage variant. In those variants, only covered elements (nodes, edges) are counted. These variants correspond to the weighted complexity metrics of D4.02 Section 6, i.e., complexity metrics computed on weighted representations of the programs in which each node or edge has a so-called simplification weight determined on the basis of (a) semantic relevance, (b) intra-execution variability, (c) the attack step in an attack model (i.e., transition in a Petri Net) for which the metric is computed, and for which it is known whether or not the attacker has already been able to collect a trace or not.

4.4 Evolution of the Metrics

During year 2, there were several occasions at which we considered adapting the proposed metrics of D4.02. We also presented the overall approach to experts in the software protection domain on several occasions:

- The project coordinator presented the approach at the invitation-only ARO workshop co-located with the ACM CCS conference (Nov 2014).

- The project coordinator gave a keynote presentation on the approach at the SPRO workshop organized by the consortium in May 2015 and co-located with ICSE 2015.
- The project coordinator presented the approach to the Scientific and Industrial Advisory Boards during their meetings in May 2015.
- The project coordinator presented the approach to external experts from NAGRA (physical meeting, Mar 2015) and to external experts from SFNT & GTO (conf call, June 2015).
- The consortium submitted a paper on the approach to the IEEE Security & Privacy special issue call for papers on the topic of Economics of Cybersecurity in January 2015.

The four live presentations were followed by extensive discussion on the challenges and opportunities of the proposed approach. At all these occasions, the message was that this approach seemed promising, albeit very challenging. Although the submitted paper was not accepted for publication, the approach itself was not criticized by the reviewers, only the lack of maturity and experimental confirmation were mentioned as reasons not to accept the paper in the special issue call for papers.
We therefore decided to keep the approach as it was presented in M12 for the time being, and first try to gain some experience with it and new insights. Now that the necessary tool support is ready, we hope to gain this experience soon.

5 Security Evaluation

Section authors:
Paolo Falcarin, Elena Gómez-Martínez, Gaofeng Zhang (UEL)

The Software Protection Assessment (SPA) tool has been developed at UEL with the purpose of evaluating the strength of a particular configuration of protections applied to a software application.
The ACTC (ASPIRE Compiler Tool Chain) is used to build a protected application by applying different ASPIRE protections (at source code level and/or binary level). Each protection can offer many configuration parameters, each one having a set of possible values, and during the build process each protection can consume one or more source code annotations.
The main goal of the ASPIRE Decision Support System (ADSS) is to search for the best ACTC configuration of such protections and parameters, the one that maximizes the protection of the code. This is a complex search problems, and the ADSS might implement different heuristics to search the problem space for the optimal configuration among all the possible ACTC configurations.
Whatever search algorithm is used by the ADSS to search for the optimum, it will need to compare the strength of two or more ACTC configurations, thus we need to define a *Protection Fitness (PF)* Function to measure the strength of an ACTC protection configuration. In the ASPIRE Security Model we decided to implement such PF function by estimating the extra effort necessary to perform the attacks modelled with Petri Nets.
The *SPA client* utilizes the SPA tool to calculate the *Protection Fitness* of the current ACTC configuration, i.e. an estimation of its level of protection, with the goal of helping the ADSS in searching for the best ACTC configuration of protections. The SPA has been designed as an independent tool which can be used by a human user or by a software client:

1. The *SPA client* is any software agent (like the ADSS) that can invoke the Java API of the SPA tool to calculate the level of protection of an ACTC configuration.
2. The *SPA user* is the actual security expert using the SPA's Extended PN Editor, to create models and assess their protection; the *SPA user* extends the capabilities of the SPA client as it can use the PN editor to insert models and then trigger the same back-end logic implemented by the SPA tool via its Java API.

In a Petri Net attack model, each transition is an attack step the attacker has to perform and the *SPA user* can use the SPA tool to define which code metrics are related to each attack step; moreover the *SPA user* can specify a linear combination of such metrics. However, as we aim at measuring the extra-effort introduced by a set of protections identified by an ACTC configuration, we cannot consider the absolute values of such metrics, but we have to consider their potency, i.e. the ratio between the metric measured after the protections have been applied and the same metric measured on the clear code before applying any protection. Moreover metrics can be combined with sign: positive if the attack step complexity increases as the metric increase, negative if the complexity increases when a metric decreases.
The SPA tool is an Eclipse plugin that implements the *Protection Fitness* function to estimate the strength of the ACTC configuration with respect to the known related attacks stored in the ASPIRE Knowledge Base (AKB) and the metrics associated by the user to each of its attack steps.
Figure 15 describes how the SPA interacts with the other main components of the ASPIRE architecture. The SPA collects the attack model data from the AKB in the ADSS, and obtains the metrics files, the log files and the version built data from the ACTC. After that, the SPA tool can be called by the ADSS (through the SPA plugin Java API) or by a user (through the SPA plugin GUI) to provide the assessment results of protection configurations under software attacks for the protection optimization of ADSS.

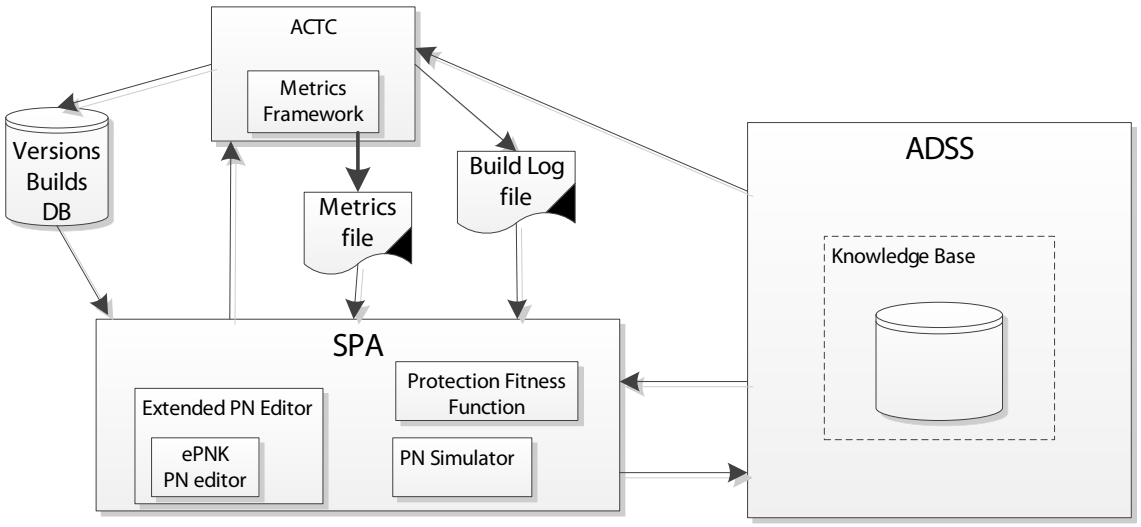


Figure 15: The Software Protection Assessment tool: ACTC and ADSS dependencies.

The SPA tool will load once the attacks and all the related useful information from the AKB through the ADSS interface: for example, the AKB contains information about which protection affects which metrics. The user must provide the information of which metric might be relevant for measuring the effort in performing a particular attack step: for this task, the user currently uses the PN editor included in the SPA tool and these data are stored in a local Petri Net Markup Language (PNML) file[5]. When the ADSS invokes the SPA tool to compute the overall metrics the (metric, attack step) associations stored in the file are used; in future versions we might consider to allow the editor to store these (metric, attack step) associations in the AKB. During the ADSS search for the optimum, the SPA protection assessment will receive one ACTC configuration, and it will invoke the ACTC to build the version of the protected application, in order to calculate the related binary code metrics, or it will load such metrics from the internal build versions database, in case such configuration has been already built during the previous searches. The attacker's additional effort (due to the applied protections configuration) for each attack step can be estimated by calculating the Protection Fitness function with a linear combination formula of the associated code metrics: this formula is more precisely described in the next section.

[5]PNML is the de facto standard format to represent Petri nets.

The SPA user can interact by means of two main phases, the setup phase and the assessment phase. In the SPA setup phase:

1. The SPA user can edit Petri Nets attack models and store them in PNML files (or in the format used by the AKB).
2. The SPA user can set the location of the software application to be assessed (or this can be sent as parameter by the ADSS).
3. The SPA user can associate any combination of metrics (from a list of metrics currently implemented by the metrics framework) to any attack step, and can edit the weights used for calculate such linear combination, for each attack step.
4. The relationship between protections and metrics is loaded once from the AKB;
5. The metrics on the software application without protections are calculated;
6. The attack paths subsumed by the Petri Net are calculated by the SPA tool (or they can be loaded by the AKB).

In the Assessment phase:

1. The SPA user chooses an ACTC protection configuration and runs the ACTC to build the software application and generate the corresponding metrics set.
2. The SPA user runs the *Protection Fitness* function which provides a quick estimation of the overall additional effort for each attack path subsumed by the Petri Net (or provided at setup time by the ADSS): the minimum additional effort calculated by the Protection fitness function can be considered an estimation of the additional effort introduced by the ACTC configuration in input.
3. In the case of random decision points in the PN diagram or in case of incomplete data (i.e. attack steps for the identified metric is not implemented), then random variables can be used to replace the missing data. A random number can be picked out of a selected range for each random variable and the PN simulator can be run to calculate the expected value and variance of the overall additional effort taking into account all the random variables.
4. The SPA user might want to run the simulator and/or the effort estimation algorithm on different versions of the code stored in the versions builds database.

The following sections provide details of the extended PN editor for attacks and metrics formula editing, the PN simulator, and some experiments with the metrics and effort estimation calculated on some simple examples.

5.1 Extended Petri Net based Editor for Protection Assessment

The Extended Petri Net (PN) editor is the tool to model software attacks. So, users can create PN-based attack models to visually describe software attacks, edit transitions' information for protection assessment, and export such models in the standard PNML file format. This file will be converted into the format used in the AKB (ASPIRE Knowledge Base).
The Extended PN editor is an Eclipse[6] plug-in that reuses an existing open-source editor for Petri Nets with Discrete Variables (PNDV), named ePNK[7]. This open-source editor is an Eclipse plug-in designed to create standard-compliant PNML models[8]. PNs with Discrete Variables (PNDVs) are a more recent PN extension that adds modelling convenience and compactness to PNs, while at

[6] `https://www.eclipse.org/`
[7] `http://www.imm.dtu.dk/~ekki/projects/ePNK/`
[8] PNML is the ISO standard Petri Net file format based on XML for general purposes, specified by ISO/IEC 15909-2.

the same time ensuring that most of the mathematical properties of PNs are still valid [3]. This model is a PN extended with a set of finite global integer variables, used in pre-conditions, which are saved on transitions: this type of extension better matches the requirements of ASPIRE security modelling.
In fact, all the information used by the attackers can be decomposed and mapped to a set of global integer variables. For example, when looking for a cryptographic key into a binary file, the attacker usually needs to identify some areas of code worth further investigation. Such intermediate knowledge could be represented with an array of code areas, where each code area is represented by a couple of integer numbers. These numbers represent the initial offset and final offset with respect to the base address of the binary code.
Hence, to use PNDVs for assessing software protection, we added the transition editing function in our tool, which allows users to edit information for each attack step (transition). In this way, the user choose which metrics can be considered good indicators of the complexity in performing the attack step, and how to make a linear combination of such metrics, by entering the weights for each metric potency.
As it can be observed in Figure 16, our extended PN editor provides extended Place/Transition (P/T) editors allowing the SPA user to type formulas and condition to each place and transition and to save it within the PNML file. The SPA implements two protection assessment methods: the protection fitness and the PN simulator. This editor is a supplementary tool for SPA users, which can create and modify PN attack models manually and then store them locally in PNML files or decide to upload such models on the AKB.

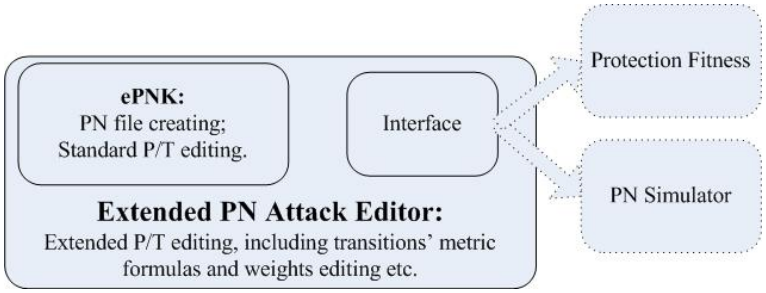


Figure 16: Extended PN editor based on ePNK.

This extended PN editor tool has two main functionalities: PN model editing and extended PN editing (with Transition Information Editing for Protection Assessment).

5.1.1 Petri Net Model Editing

The function of PN model editing in our Extended PN editor is provided by ePNK, including two parts: PN file creation and standard Place/Transition editing.

Creating a PN file. The ePNK editor can be used to visually create the PN attack models and save them as PNML standard files (ISO/IEC 15909-2). So, users can create a PNML file as a regular XML file in Eclipse. In this way, the users can use the SPA plugin as an independent tool with which they can create attack models in PNML files, associate metrics to attack steps and manually estimate the software protection fitness by interacting with the ACTC and the Metrics framework. In a the more advanced scenario, the ADSS can automatically invoke the API of the SPA plugin, to use its Protection Fitness function during the search for the optimal ACTC configuration, combined with all the additional information stored in the AKB. More in general the PN editor could be used as a user-friendly interface to insert attacks into the AKB, instead of editing the AKB directly; the PNML model edited via the PN editor can be eventually translated to AKB, but this engineering task will not be developed in the project.

Editing Places and Transitions. As introduced in the ASPIRE deliverable D4.01, software attack steps can be described by places and transitions. Therefore, we need to edit them to modify attack steps. Based on ePNK, there are two modes to edit places and transitions: the text editing and graphical editing.
The first one is the text editing to modify directly places, transitions, and arcs in the XML format. If users are very confident in PNML files and XML formats, it is an available path to edit PN models on software attacks. However, the easiest way to edit PN models is by means of the graphical editing.
Figure 17 is a screenshot depicting our editor tool. The central view is the graphical editing space with the "Palette" menu to edit places, transition and arcs in the Extended PN-based attack models. For example, users can add places, transitions and arcs by clicking the corresponding options in the "Palette" panel.

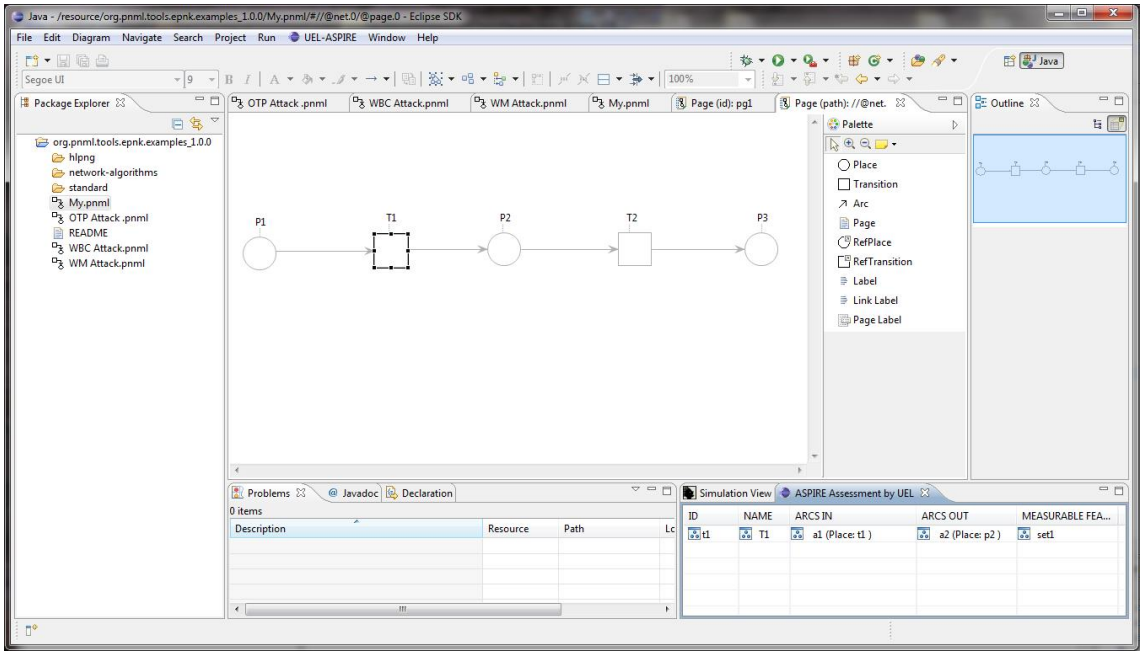


Figure 17: The graphical editing for PN models.

Besides, it also includes "Outline" view, "Property" view, and "UEL-ASPIRE" menu to start assessing (including run the simulator), as can be observed in Figure 17. The "Outline" view provides a schematic diagram about the PN model in the edition process. The "Property" view helps to display some information on selected places or transitions. The "UEL-ASPIRE" menu provides the triggers to run diverse assessment methods.
In this editor tool, SPA users can edit PN models to represent software attacks and add weights to the metric formulas attached to the attack steps (transitions). The editor features are introduced in the next section.

5.1.2 Transition Information Editor for Protection Assessment

The Transition Information Editor for protection assessment allows entering two kinds of information needed to be edited in PN based attack models. The first one is the relevance or weight of each attack step; the other one is the formula with the involved metrics considered good effort indicators for a specific attack step. The editor process them as string data types.
To edit these two kinds of information, users can create string data type for each transition using the "Label" option of the "Palette" menu. Figure 18 illustrates an OTP attack model with transition information for protection assessment taken from ASPIRE document D4.02. For this attack model, SPA users can set all transition information based on their expertise and knowledge. Note that the

values on this transition information of the present subsection are only used as example.

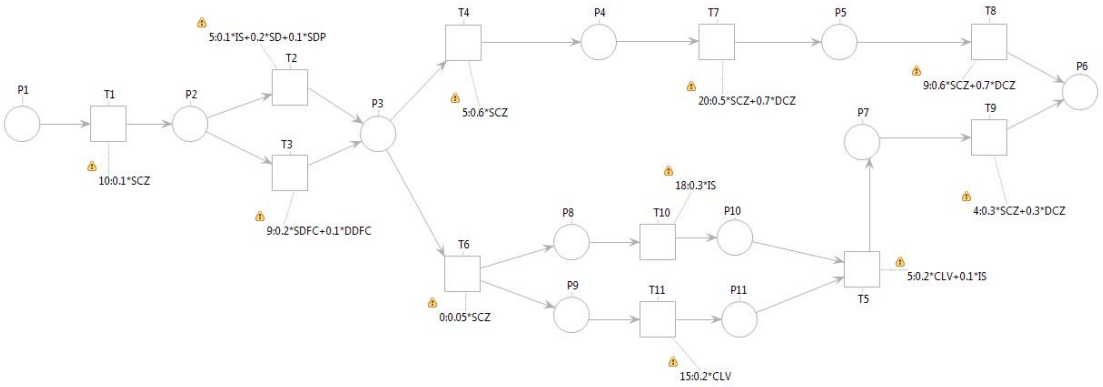


Figure 18: PN models with transition information for protection assessment.

Let us take transition T_3 in Figure 18 contains the string label "$9 : 0.2 * SDFC + 0.1 * DDFC$". The "$SDFC$" and "$DDFC$" are two software metrics taken from a list of the available metrics implemented by the Metric Framework: section 5.4 will discuss the software metrics implemented. As it can be observed, this string label includes the *Relevance* (weight of the transition) T_3, which is "9" and means the significance of this transition in the whole attack, as explained later in Section 5.2.1. The metric formula of the transition T_3 is "$0.2 * SDFC + 0.1 * DDFC$", being "0.2" and "0.1" are the weights for the metrics "$SDFC$" and "$DDFC$", respectively, in transition T_3. The labels "$SDFC$" and "$DDFC$" will be actually represented by the corresponding metric potency, as defined in Section 5.2.1.

5.1.3 Property View in the Graphical Editing Model

In our extended editor, to facilitate the edition of a P/T and its transition information, the property view has been created to display some information about selected places or transition as shown Figure19.

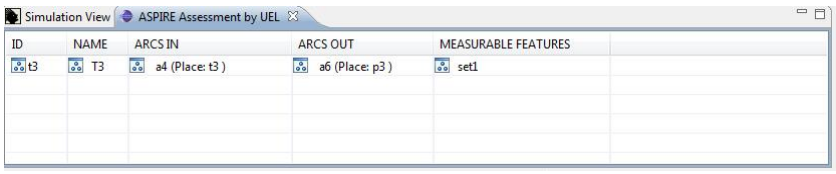


Figure 19: Property view of the editor tool.

In summary, the extended editor provides a tool that allows SPA users to edit PN based attack models by editing places and transitions that represent software attacks and attach information to transitions (attack steps), such as relevance and metric formula of each transition. The editor can also be used to invoke the different protection assessment methods, such as the protection fitness function and the PN simulator described in the next sections.

5.2 Protection Fitness Function

In this section, we will introduce the protection fitness function based on the transition information for protection assessment.

5.2.1 Transition Information for Protection Assessment

Weight of a Transition. In PN based attack models, each transition can have different relevance (weight) depending on the attacker skills. For instance, if attackers spend more time in one transition (attack step) than others. So, the relevance of this transition (attack step) should be higher, since it represents its significance within the attack. In the current implementation of the SPA tool the relevance parameter for each transition is set to 1 by default, but the SPA user can use this multiplier of the corresponding metric formula to represent the attacker skills for a particular attack step; for example the SPA user might want to set the relevance (weight) value close to zero for expert attackers for attack steps which can be automated by using tools known by the expert attacker; instead a non-expert attacker might be better represented by a value greater than 1 as their limited knowledge contributes to the increase of the effort in performing the attack: we will explore how to map this additional parameter to the attacker profile in the next year of the project. The transitions' weights are defined with the formula shown in the vector 1.

$$W = \{w_1, w_2, \ldots, w_t\} \tag{1}$$

being t is the number of transitions (attack steps) in the Petri Net model. For instance, we can find out that the weight of T_3 is $w_3 = 9$ in Figure 18.

The Metric Formula of Transition. In PN based attack models, each transition has a different metric formula to represent a linear combination of the potency of the metrics considered good effort indicators for that specific attack step. This formula has similarly been presented in other ASPIRE documents, such as Section 6 in deliverable D4.02. In this section, we define all metric formulas as follows:

$$F = \{f_1, f_2, \ldots, f_t\} \tag{2}$$

where t is the number of transitions of the Petri Net model.
Hence, the specific metrics formulas are:

$$f_i(PC_k) = w_{i,1} \times \frac{m_1^a(PC_k)}{m_1^b(PC_k)} + w_{i,2} \times \frac{m_2^a(PC_k)}{m_2^b(PC_k)} + \ldots = \sum_{j=1}^{p} w_{i,j} \times \frac{m_j^a(PC_k)}{m_j^b(PC_k)} \tag{3}$$

In this equation, $m_j^a(PC_k)$ and $m_j^b(PC_k)$ are two parts of software metrics, as introduced in equations 4 and 5. PC_k is the Protection Configuration which is under assessment, $m_a(PC_k)$ is the software metric values AFTER protection configuration: PC_k being executed, and $m_b(PC_k)$ is the software metric values BEFORE protection configuration: PC_k being executed. p is the number of software metrics being evaluated.

$$M_a(PC_k) = \{m_1^a(PC_k), m_2^a(PC_k), \ldots, m_p^a(PC_k)\} \tag{4}$$

$$M_b(PC_k) = \{m_1^b(PC_k), m_2^b(PC_k), \ldots, m_p^b(PC_k)\} \tag{5}$$

The $w_{i,j}$ parameters in metric formulas, are stored in the Weight Matrix $(t \times p)$ for metrics in each transition:

$$WM = \begin{pmatrix} w_{1,1} & \cdots & w_{1,p} \\ \vdots & \ddots & \vdots \\ w_{t,1} & \cdots & w_{t,p} \end{pmatrix} = (w_1, \ldots, w_p) \tag{6}$$

Besides, if the value of $w_{i,j}$ is zero, it means that the related software metrics: $m_j^a(PC_k)$ and $m_j^b(PC_k)$ have no influences on this transition: T_i.
In ASPIRE, currently, all weights and metric formulas of transitions from PN based attack models will be decided by security experts, based on their personal knowledge on the specific software

attacks. They can be stored locally or in AKB (in future) to support the reuse and the automatic assessment. If not specified by the user, all the weights of the chosen metrics are set at 1 by default. For example, in case of the metric formula: $f_3 = 0.2 * SDFC + 0.1 * DDFC$, there are only two metrics: SDFC and DDFC in this metric formula, which means that all other metrics' weights $w_{i,j}$ are zero, while for these two metrics in this transition, there are two weight values: "0.2" and "0.1", respectively.

5.2.2 Protection Fitness Function Method

In this section, we will introduce the protection fitness function, on the basis of the transition information for protection assessment and software metrics from software needed to be assessed. Based on equations 2-6, this method generates attack paths in this attack model (depending on the attack model input), calculates the potency for each attack path, and obtains the final assessing result.

Attack Path Generation. When the SPA plugin is used as an standalone tool, we generate all possible attack paths from the Petri Net attack model. When the SPA tool is used within the ADSS, the attack paths are computed by the ADSS and loaded from the AKB. In both cases, we generate a Path Matrix (PM) to store attack paths.
The value of each item in the PM is 1 or 0. For example, $pm_{i,j} = 0$ means that attack step T_j is not in the path P_i; otherwise, $pm_{i,j} = 1$ means that attack step T_j is in the path P_i.

$$PM = \begin{pmatrix} pm_{1,1} & \cdots & pm_{1,n} \\ \vdots & \ddots & \vdots \\ pm_{t,1} & \cdots & pm_{t,n} \end{pmatrix} \tag{7}$$

The Effort Calculation for One Path is based on Equation 1 and Equation 3 to compute the Effort Increase (ΔE) for each transition.

$$\Delta E(T_i, PC_k) = w_i \times f_i(PC_k) \tag{8}$$

Then, for one attack path, we can do the weighted sum for all attack steps included in order to obtain the ΔE for one attack path by Equation 9.

$$\Delta E(P_i, PC_k) = \sum_{j=1}^{n} (\Delta E(T_i, PC_k) \times pm_{i,j})) \tag{9}$$

Protection Assessment. To consider the worst condition on protection, the lowest one of all ΔE values from these attack paths represents the general Delta Effort of the whole Petri Net attack model under the specific protection configuration, as indicated in Equation 10. In case of two protection configurations to be compared, we should compute the two assessment results for both protection methods, respectively: in this case, the better protection method will be the one with a higher ΔE.

$$ProtectionFitness(PC_k) = min(\sum_{i=1}^{t} (\Delta E(P_i, PC_k)) \tag{10}$$

5.3 PN Simulator

For protection assessment, another tool is the PN simulator to compute the Monte Carlo simulation of PN attack models. In this section, this simulator is designed to run protection assessments based on Monte Carlo simulation to be used to extend the Protection Fitness function when random variables can be used to represent missing data, and choices in the Petri Net models.

The simulator will be introduced in two parts: Single Attack Process simulation and Monte Carlo simulation. The first part focuses on the execution of the single attack simulation for one specific attack process. The attack can be successful or unsuccessful, so the result of this simulation will be TRUE or FALSE. The input for this part includes PN attack models and simulation information (attacker effort and effort consumption, which are random variables). For example, these effort consumptions are decided by specific protection configurations for assessing the potency of protection configuration. The second part is the Monte Carlo simulation, which can execute the previous single attack process simulation repeatedly and determine the probability of successful attacking as the result of this simulation, based on the TRUE/FALSE results from the repeated single attack process simulation. The greater probability means lower potency for this protection configuration.

5.3.1 Single Attack Process Simulation

In this section, we introduce the single attack process simulation, including two parts: simulation information and single attack process simulation.

Simulation Information Based on the PN attack models, there are two kinds of simulation information for PN simulator: Effort Consumption (*EC*) and Attacker Effort (*AE*).

- EC represents the Effort Consumption. It is a finite set of attacker's effort consumed at each transition, where $EC = \{ec_0, \ldots, ec_t\}$.
- AE represents the Attacker Effort. It is a finite set of attacker's effort in each state in P, where $AE = \{ae_0, \ldots, ae_n\}$. And n is the number of places in the PN model. Attackers have the capability including resources and skills to execute attacks on protected or unprotected software. This "capability" is AE. And this capability will be "consumed" in transitions of attack processes via EC in attack simulations.

Effort Consumption—EC and ec_i can be described by the uniform distribution as random variables. For each ec_i, a Maximum boundary $-Max_i$ and a Minimum boundary $-Min_i$ determine the random variable by the uniform distribution, by Equation 11.

$$EC = \{ec_0, \ldots, ec_i, \ldots, ec_t\}, ec_i = fec(Min_i, Max_i), i \in [0, t] \tag{11}$$

In Equation 11, $fec()$ represents the sampling process of the uniform distribution with two boundaries: Min_i and Max_i. For example, T0 in the OTP attack model as described in Figure 18 is to "Identify the PIN check portion of the code". Both Max_0 and Min_0 can be pre-set in the attack modelling, based on real attack data provided by security experts in industry. After that, ec_0 is the random variable with the uniform distribution and two boundaries: Max_0 and Min_0. Max_0 and Min_0 can be increased due to the fact that some protections have been applied: for example, when some software protection methods increase the code size or complexity, this can make the T0 attack step more difficult, which will change the uniform distribution for ec_0 with Max_0 and Min_0. These methods could be specific PSs to change ec_0. The relations between methods and transitions are decided by security experts, too.
Another item is AE, which represents the current effort of the attacker in the state of this attack process. And AE can be described by equation 12. In this equation, ac_0 is the attacker effort before attack processes, as a random variable with a normal distribution. In the simulation process, this random variable has been set by security experts.

$$AE = \{ac_0, \ldots, ac_i, \ldots, ac_n\} \tag{12}$$

Single Attack Process Simulation The general process of Single Attack Process Simulation (*SAPS*) works as follows: in a PN model (as a Directed Acyclic Graph), one attacker will try to find a path from the starting state to the final state. If he/she finds one, the result of this *SAPS* is TRUE; otherwise, it is FALSE. It is a route searching process in a directed acyclic graph.
In *SAPS*, in each node, e.g. transition, the Passing Probability (*PP*) is used to control the probability that the attacker can complete this transition and reach the next state. We can set them by Equation 13. Passing Probability (*PP*) is a finite set for each transition in T, and $pp_i \in PP, i \in [0, t]$.

$$pp_i = \left\{ \begin{array}{ll} 0, & ae_{CUR} < ec_i \\ \tanh(\frac{ae_{CUR}}{ec_i - 1}), & ae_{CUR} \geq ec_i \end{array} \right\} i \in [0, t] \tag{13}$$

In equation 13, ec_i comes from equation 11, which is the effort consumption for each attack step. And ae_{CUR} comes from equation 12, which is the current attacker effort in one attack simulation process. If ae_{CUR} is smaller than ec_i, the probability is zero, which means that the current attacker effort is too low to complete this attack step. Otherwise, if ae_{CUR} is not smaller than ec_i, the passing probability is supported by the hyperbolic tangent function: $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. It is monotonically increasing and in the range of $[0, 1)$.

$$AE_{NEW} = ae_{CUR} - ec_i \tag{14}$$

In transition Ti, on the probability of pp_i, equation 14 will be executed, which means that the attacker passes this transition. Otherwise, the attacker needs to go back to the previous state to find out other paths to reach the final state.

5.3.2 Monte Carlo Simulation

The Monte Carlo method can be used to manage *SAPS* and provide a Monte Carlo based attack simulation to assess software protection, as depicted in Figure 20. The central component is the *SAPS*. To run *SAPS*, we need to do an initialisation (build the PN model with EC and AE). The result of each *SAPS* is a boolean value. Then, the Monte Carlo method executes the *SAPS* several times. At last, the simulation provides a probability of attack success (the ratio of *SAPS*s with TRUE in all *SAPS*s).

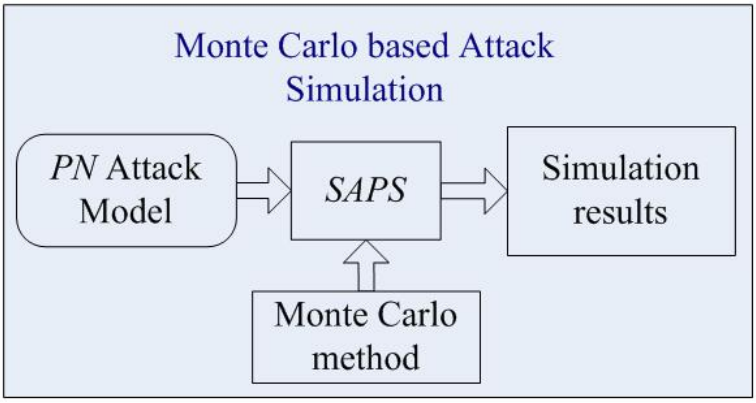


Figure 20: Monte Carlo based Attack Simulation.

In future, the simulator is planned to be updated to cooperate with the protection fitness function introduced in Section 5.2: in the process of protection fitness function, if some necessary transition information is missing (such as weights of attack steps, or the metrics formula of attack steps), which means the protection fitness function cannot be executed, the PN simulator can be triggered to use the Maximum boundary $-Max_i$ and the Minimum boundary $-Min_i$ for each ec_i of the EC to execute the protection assessment.
In other words, the simulator will be used to aid the protection fitness function to integrate ACTC and ADSS together from the perspective of protection assessment in ASPIRE project.

5.4 Obtaining Metrics with ACTC

Software metrics measure quality of software from a quantitative viewpoint. Specifically, security metrics allow to measure security features of a software protection against attackers. In the absence of generally accepted metrics for software protection algorithms, ASPIRE project proposes concrete metrics in order to evaluate software protection tools.
The first proposal of the measurable features and their corresponding security metrics was presented in the deliverable D4.02. This section describes the subset of metrics resulting from the execution of the ASPIRE Compiler Tool Chain, named ACTC, and will be used to evaluate and assess protection tools.
Complexity metrics can be static or dynamic, being obtained in different phases of the ACTC. So, static metrics are resulting from tool chain when the protected application or library, called d.out and libd.so respectively, is obtained in BC05 (see Section 9 in deliverable D5.01). Dynamic metrics are obtained after running the aforementioned protected application, for instance, on an ARM board. As commented in the deliverable D4.02, dynamic metrics depend on the program inputs that are selected to execute and trace the programs. These metrics that we can currently obtained after executing the ACTC are summarized in Table 1. These metrics do not take into account if the protection tool is applied to the entire source code or only a part of it. Moreover, remark that these metrics do not correspond with those ones supported in Section 4.2, since this is a preliminary version for the security evaluation.

	Description	**Static/Dynamic**
NB	Number of bytes in the executable measured by means of the following command line: `wc -c`	Static
INS	Number of instructions	Static
EDG	Number of edges	Static
IND	Number of index edges	Static
SRC	Number of source operations	Static
DST	Number of destination operations	Static
EXE	Number of executed instructions	Dynamic
SOP	Number of source register operands	Dynamic
DOP	Number of destination register operands	Dynamic
JMP	Number of computed jumped instructions	Dynamic

Table 1: Metrics resulting from the ACTC execution.

The metrics described in Table 4.2 allow us to calculate the metrics supported by Diablo in Section 4.2:

$$SPS_{source} = SRC + SOP \tag{15}$$

$$SPS_{dest} = DST + DOP \tag{16}$$

$$CC = EDG \tag{17}$$

$$CFIM = IND \tag{18}$$

To study these metrics, we have executed the ACTC with a simple example. With this aim, we recall a software attack model taken from literature. Wang et al. [4] use a Serial Number Certification program to evaluate software protection. So, they protect this example with code obfuscation techniques by means of a protection tool. Moreover, they model the attack against this example using a Software Attack Model based on Marked Petri Net (SAMMPN). For the sake of clarification, we include the Petri net representing the software attack process, depicted in Figure 21, as

well the meaning of each Place/Transition, described in Table 2. As the authors pointed out and it can be observed in Figure 21, there are two path for the attack: i) the attack destroys the Serial Number checking by modifying the key instruction, and ii) the attack destroys the Serial Number checking by adding new codes into the program.

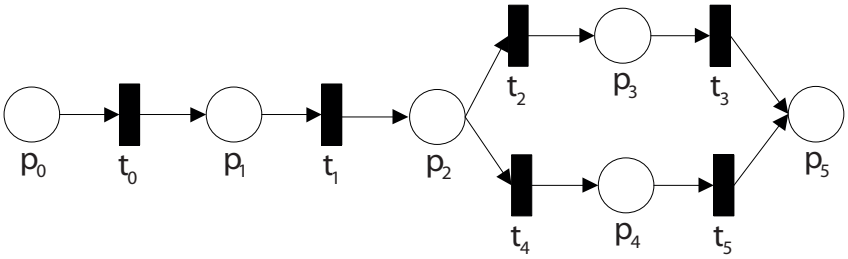


Figure 21: SAMMPNN taken from Wang et al. [4].

P/T	Meaning
p_0	Get the program.
p_1	Get the assembly code.
p_2	Get the Basic Blocks.
p_3	Get the performed Basic Blocks.
p_4	Get the Key Block.
p_5	Get the cracked program, which always succeed.
t_0	Disassemble the program.
t_1	Split the code into Basic Blocks, mark the Success Block and the Fail Block.
t_2	Debug the program, and mark the performed Basic Blocks.
t_3	According to the Running Sequence of Success and performed Basic Blocks, locate the Key Block and amend Key instruction in it.
t_4	Monitor the change of memory while performing the program and stop when the address of Fail block was generated and mark the stopped blocks as the Key Block.
t_5	Add New Codes into redundant space of the program: amend Key Blocks so that it always jumps to New Codes.

Table 2: Meaning of States(P) and Techniques (T) of Figure 21.

We have implemented a simplistic version of a serial number generator, as well as the corresponding certification program. We protect the certification program by means of ACTC under different options. Table 3 summarizes the obtained metrics. Note that we compute these metrics considering that a particular protection tool is applied to the entire source code.
Column *Original* of Table 3 shows the size of the original binary, i.e., the executable compiled and linked using `gcc` command. Obviously, this is an unprotected program. Columns *a.out*, *c.out* and *d.out* detail the size of the executable generated with different steps of the ACTC (version 1.1.0), where WBC was still implemented by means of simple XOR-ing instead of proper AES. It is for that reason that the WBC-protected binaries are smaller than the unprotected version: a complex AES was replaced by a simple XOR algorithm. For more information about that ACTC version, we refer to the deliverables of WP5. Sizes are always measured using the following command line: `wc -c *.out`.
Once we executed the ACTC and we obtained the metrics, the next step is to map these metrics into the PN. Wang et al. [4] propose six grades to classify each attack step: Grade A is for attack

Protection		Size (bytes)				Static Metrics				
		Original	a.out	c.out	d.out	INS	EDG	IND	SRC	DST
None (original)		8768	-	-	-	-	-	-	-	-
Static linker	Plain	-	709802	765726	483616	93670	40225	819	111747	130072
	WBC	-	708495	764403	483576	93642	40204	819	111696	129981
	Obf. merge	-	709858	765782	483680	93687	40227	819	111764	130088
	Obf. xor	-	709782	765690	483624	93673	40227	819	111748	130076
Dynamic linker	Plain	-	12254	69536	30987	5970	1645	120	7177	7207
	WBC	-	10010	68225	30891	5942	1626	120	7126	7116
	Obf. merge	-	12306	69596	31059	5987	1647	120	7194	7223
	Obf. xor	-	12226	69516	31003	5973	1647	120	7178	7211

Table 3: Metrics obtained with Serial Certification example.

techniques that can be performed automatically and Grade F is for those techniques that attackers perform manually, i.e., without any assistant tools. We reinterpret this scale in order to predict what metrics should be involved in each attack step, see Table 4.

Transition	Grade	Involved metrics
t_0	Grade A	SIZ
t_1	Grade B	INS, EDG, IND
t_2	Grade B	EXE
t_3	Grade C	EXE(t_3) - EXE(t_2)
t_4	Grade E	EXE(t_3) - EXE(t_2)
t_5	Grade D	SIZ

Table 4: Metrics involved in each attack step.

In the next year, we will study how to improve the usage of metrics shown in Table 4 in our assessment tool described in Section 5.1 and 5.3.
For example, once a set of ACTC configurations have been run to build different application versions (along with the corresponding set of metrics), this dataset can be then used to further refine the metrics formula in each attack step. The main problem of a linear combination of metric potencies is how to set the weights for each metric in order to be able to normalize each metric contribution to the overall sum. Once a set of metrics has been calculated, the maximum value of each metric can be stored and used as denominator to achieve normalization of each metric potency: indeed the weight of particular metric could be assigned a value of *1/max*, where *max* is the maximum value calculated so far for that metric. In this way the SPA tool will be able to implement machine learning techniques on this dataset with different goals:

- Refine the maximum metric value to be used for normalization of weights.
- Perform ANOVA and factor analysis on the matrix protection/metrics to identify which subset of protections are actually the main factors of variance for each metrics; this information can be used by the SPA plugin to implement heuristics for searching for an optimal ACTC configuration by prioritizing the protections causing more metric variability in the search the identified factors, in case the full search by the ADSS will take too much time to converge.
- Use such statistical tests to verify whether the relationship between protections and metrics is actually confirmed or rejected by data analysis.

Part III
Experiments

The objective of the experiments is to investigate the level of protection offered by ASPIRE protections from and empirical point of view. They are divided in experiments with academic participants and experiments with industrial participants.
In the experiments with academic participants, the experimental setting is represented by an artificial environment (i.e., in vitro experiment) where the experimenter controls and objectively measures all the relevant variables. This allows to apply statistical analysis to elaborate objective observations. Academic participants are involved in multiple rounds of experiments to test ASPIRE protection, but each of them in isolation.
In experiments with industrial participants, conversely, we intent to assess ASPIRE protection in a more realistic setting, with many protections applied to real code that will be attacked by real and expert hackers. This setting does not allow to apply statistical test and only subjective evaluations will be formulated, however they will be more realistic and representative of a real attack scenario.

6 Data obfuscation experiment

Section authors:
Mariano Ceccato, Paolo Tonella (FBK), Marco Torchiano (POLITO),

Goal	Analyze the ability of data obfuscation to protect sensitive data inside the code
Treatments	T0 = clear code; T1 = RNC data obfuscation; T2 = XOR data obfuscation; T3 = var merge data obfuscation
RQ1	How effective is data obfuscation in protecting data inside the code as compared to the clear code?
RQ2	What data obfuscation technique is most effective between T1/2/3 in protecting data inside the code?
Subjects	Students from FBK, UEL, POLITO and UGent (working on the binary code)
Objects	P1=Lottery, P2=Lotto (C code): programs for the extraction of lottery/lotto numbers
Tasks	Force the program to extract only numbers between 1 and 20; leak the winning sequence from the program
Metrics	Success rate; time to mount a successful attack
Design	Lab1: P1-T0; P1-T1/2/3 (Repl 1/2/3); P2-T1/2/3 (Repl 1/2/3); P2-T0 Lab2: P2-T1/2/3 (Repl 1/2/3); P2-T0; P1-T0; P1-T1/2/3 (Repl 1/2/3)

Table 5: Data obfuscation experiment

Table 5 provides a schematic overview of the data obfuscation experiment. The *goal* of this experiment is to analyse the degree of protection offered by the ASPIRE data obfuscation techniques, when these are applied to some sensitive data that reside inside the source code. Specifically, three different data obfuscation techniques have been investigated in this experiment:

Residue Number Coding (RNC): numeric values are protected by encoding them as a tuple of integers computed as the respective residues over a tuple of moduli.

XOR: numeric values are masked through the XOR boolean operation.

Variable merge: different variables are merged into a single variable.

In each replication of the experiment, one of these three data obfuscation technique is evaluated in comparison with the clear code. This allows for a direct measurement of the reduced success rate/increased attack effort associated with each protection, but it also allows for an indirect comparison among different alternative techniques, achieved through different replications that involve different protections. The replication at UGent has been carried out directly on the binary code. At UGent, the project's coordinator's research group organized an Ethical Hacking Student Work Group in the academic years 2012–2013, 2013–2014, and 2014–2015, in which students were introduced to the reverse engineering of binary code. As a result, sufficient students are available with (some) experience with binary code reverse engineering tools such as IDA Pro to participate in this experiment. None of the students had received training by the UGent with respect to obfuscations, however.
The first three replications of this experiment, conducted respectively at FBK, UEL and POLITO, evaluated treatment T1 (RNC), T2 (XOR) and T3 (Var merge) in comparison with the baseline treatment T0 (clear code). The replication at UGent evaluated T1 (RNC) against T0, with the binary code instead of the source code provided to the students.

6.1 Research questions

The experiment aims at answering the following two research questions:

- **RQ1**: How effective is data obfuscation in protecting data inside the code as compared to the clear code?
- **RQ2**: What data obfuscation technique is most effective between T1/2/3 in protecting data inside the code?

The first research question deals with the effectiveness of data obfuscation protection. Data obfuscation might result in a higher time to successfully complete an attack task or might result in the failure to complete the attack task when this is conducted on protected code. There is an economic advantage in adopting a data obfuscation protection if the probability that an attack is successfully completed within a limited amount of time is drastically reduced by the protection. Hence, RQ1 is a key research question for the ASPIRE project.
The second research question is a comparative one. By contrasting the effectiveness of the protection across different replications of the experiment we get insights about the relative effectiveness of T1 (RNC), T2 (XOR) and T3 (Var merge).

6.2 Objects

The objects of this experiment are two C programs, Lotto and Lottery, obtained from the web and developed by third parties who are not involved in any way in the ASPIRE project.

Lotto is a C program for machines that let users play lotto. A new program is generated every week, with the winning sequence embedded in the source code. The JACKPOT is hit when all 7 numbers are matched (i.e., all six numbers plus the bonus number). The source code of Lotto consists of 234 SLoC (Source Lines Of Code), as measured by the UNIX utility `sloccount` [5].

Lottery is a C program for machines that let users play lottery. Lottery uses a random sequence of bytes (called a challenge) generated by a remote server to extract 7 numbers (without repetitions) between 1 and 39. The challenge is logged in the remote server, to allow for later anti-tampering check. A legal extraction consists of 7 numbers that match one of the

challenges stored inside the remote server. 50 extractions are repeated in 3 iterations. Lottery prints the 50×7 extracted numbers and the statistics collected in such extractions, consisting of the frequency of occurrence of numbers 1, . . ., 39 in the extractions. The source code of Lottery consists of 196 SLoC, as measured by the UNIX utility `sloccount`.

Both Lotto and Lottery have been obfuscated by means of T1 (RNC), T2 (XOR) and T3 (Var merge). The two attack tasks to be executed on the two objects are respectively:

Lotto: Determine the JACKPOT sequence, consisting of 7 winning numbers, which is embedded in the C source code (specifically, inside some variables) of the Lotto program. While executing the task, subjects are allowed to read, modify, compile, debug and execute the source code. The program should be modified so that it prints the winning sequence to the standard output as soon as the program is launched (subjects working on the binary code have been asked just to write down the winning sequence on paper).

Lottery: Modify the program Lottery so that it extracts only numbers between 1 and 20 in a legal extraction (i.e., one matching the logged challenge). When any of the 7 extracted numbers is greater than 20, the extraction is redone, by requesting a new challenge to the remote server. In fact, the extracted numbers are checked for validity against the challenge stored in the remote server. After successfully completing the task, the reported frequencies shall be equal to zero for all numbers greater than 20.

In the pilot experimental sessions conducted to fine tune the experimental material, we realised that the attack task on Lotto is substantially easier to perform than the attack task on Lottery. We think this increases the range of validity of the experimental results, since they span from simpler to more complex attack tasks.
The Powerpoint presentation used to introduce the experiment and the tasks to the subjects is provided in Appendix B.

6.3 Metrics

The two metrics collected to answer research questions RQ1 and RQ2 are:

AT: Attack time

SR: Success rate

Subjects are asked to mark down the start and end time when starting and after finishing the attack task, so one key metric collected during the experiment is the attack time (AT). Subjects are also asked to show their attack, in case they deem it successful, to the researchers, who verify if the attack was successful or not. Students were originally told they could spend two hours on the task. But since the classroom was available much longer, many of them actually tried much longer, up to 5 hours before giving up. The times after which the subjects gave up in the UGent experiment are visualized in Figure 22. As they stopped their attempt for very different reasons, it is not useful to include these times in any statistical analysis or hypothesis testing. So the times are only provided for the sake of completeness, and because they provide useful insights, albeit not backed up by statistical analysis. Metrics AT is meaningful only for successful attacks. The proportion of successful attacks provides a second metric, which complements AT, called success rate (SR). SR measures the proportion of subjects that successfully completed the attack task either on the clear code or on code protected by data obfuscation.
Based on the metrics chosen to quantify the effectiveness of the protections, we can formulate null and alternative hypotheses associated with research questions RQ1 and RQ2:

- **H1-AT$_0$**: There is no difference in AT between subjects working on obfuscated and subjects working on clear code

- **H1-SR$_0$**: There is no difference in SR between subjects working on obfuscated and subjects working on clear code
- **H2-AT$_0$**: There is no difference in AT between subjects working on code obfuscated by technique T and subjects working on code obfuscated by T′
- **H2-SR$_0$**: There is no difference in SR between subjects working on code obfuscated by technique T and subjects working on code obfuscated by T′
- **H1-AT$_a$**: Data obfuscation increases AT for subjects working on obfuscated as compared to subjects working on clear code
- **H1-SR$_a$**: Data obfuscation decreases SR for subjects working on obfuscated as compared to subjects working on clear code
- **H2-AT$_a$**: Data obfuscation technique T is more effective than T′ for what concerns AT
- **H2-SR$_a$**: Data obfuscation technique T is more effective than T′ for what concerns SR

where T and T′ indicate two different data obfuscation techniques taken from the set T1, T2, T3. In addition to the metrics AT and SR, we ask subjects to answer a pre-questionnaire and a post-questionnaire. Both are included in the instructions for the subjects provided in Appendix D. The pre-questionnaire collects information about the abilities and experience of the involved subjects. This is very important to analyse the effect of ability and experience in the successful completion of the attack tasks, either on clear or on obfuscated code. The post-questionnaire collects information about clarity and difficulty of the task, availability of sufficient time to complete it, and on the tools used and the activities carried out to complete the task.

6.4 Design

The design of this experiment is counterbalanced so as to ensure that each subject works both on clear code and on protected code, and to ensure at the same time that each subject works on different objects during different labs. In this way, learning effects that might occur across labs are to some extent compensated by the balanced design.

Lab	P1-T0	P1-T′	P2-T0	P2-T′
Lab1	G1	G2	G3	G4
Lab2	G4	G3	G2	G1

Table 6: Design for the data obfuscation experiment: each group of subjects (G1/G2/G3/G4) is assigned a different object (P1/P2) and treatment (T0/T′) in each lab (Lab1/Lab2)

Table 6 shows the counter-balanced design of the experimental sessions. Subjects are divided into four groups, G1, G2, G3 and G4. Objects (P1 = Lottery; P2 = Lotto) are provided as clear code (T0) or obfuscated code (T′). In the latter case, the obfuscation varies across replications (T′ = T1, T2 or T3 depending on the replication). In the two consecutive laboratories, each group of subjects receives both a different object and a different treatment.

6.5 Statistical analysis

The difference between the output variable (AT or SR) obtained under different treatments (clear code vs. obfuscated code) is tested using non-parametric statistical tests, assuming significance at a 95% confidence level (α=0.05); so we reject the null-hypotheses having p-value<0.05. In particular, we perform paired Wilcoxon test when applicable i.e., when subjects took part in both labs of each experiment, and non-paired Mann-Whitney test on all samples when data are not

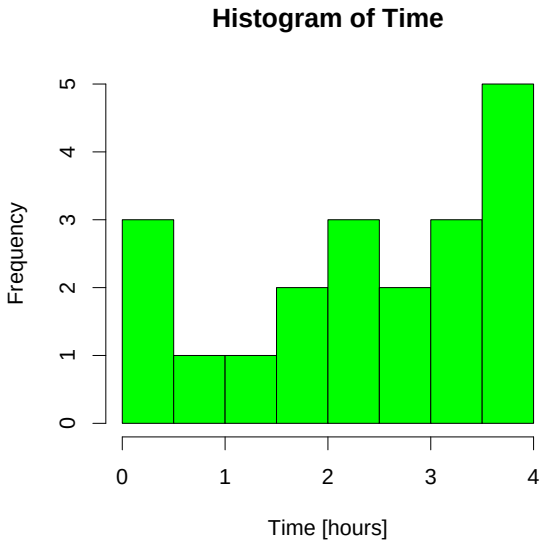


Figure 22: Time spent by UGent's subjects to work the attack task (successful and not successful, both on clear and obfuscated code).

paired because of subjects that participated just to one lab [6]. Regarding the analysis of pre and post questionnaires whose answers are on a Likert scale, we test the difference of the medians by means of the non-parametric Mann-Whitney statistical test, assuming significance at a 95% confidence level (α=0.05). For correlation analysis, the Spearman's rank correlation test [1] is used, still with significance level set to α=0.05.

6.6 Experimental results

6.6.1 UGent results

At UGent, the researches provided a warm-up exercise to student (in line with the advice obtained from the project's Scientific Advisory Board). Trying out the attacker tools on this example was voluntary, and the subjects could do it by themselves beforehand, or under guidance the hour before the first experiment session.
The binary code was generated with a recent version of gcc, without letting the compiler perform code optimization.
Figure 23 shows some descriptive statistics about the participants involved in the UGent's experiment, which was carried out on binary code. These data were collected by means of the pre-experiment questionnaire reported in Appendix C.
At UGent 10 MSc or PhD students participated to the two sessions of the experiment. Most of them had no previous experience as professional programmers. On the other hand, the majority declared two or more years of experience with the C programming language and with a C programming IDE (Integrated Development Environment), such as Eclipse or Visual Studio. Their experience with the Assembly language was a bit shorter; still, most of them have been using Assembly for at least one year. Their experience with reverse engineering of Assembly code was relatively short but significant, ranging between 6 months and one year for the majority of the subjects. All subjects declared good knowledge of the C and Assembly debuggers.
Overall, subjects participating in this experiment have little professional experience, substantial experience with C and Assembly, some previous experience with reverse engineering and they know the debugger pretty well.
Figure 24 shows the results of UGent's experiment. Histograms on the top-left show the suc-

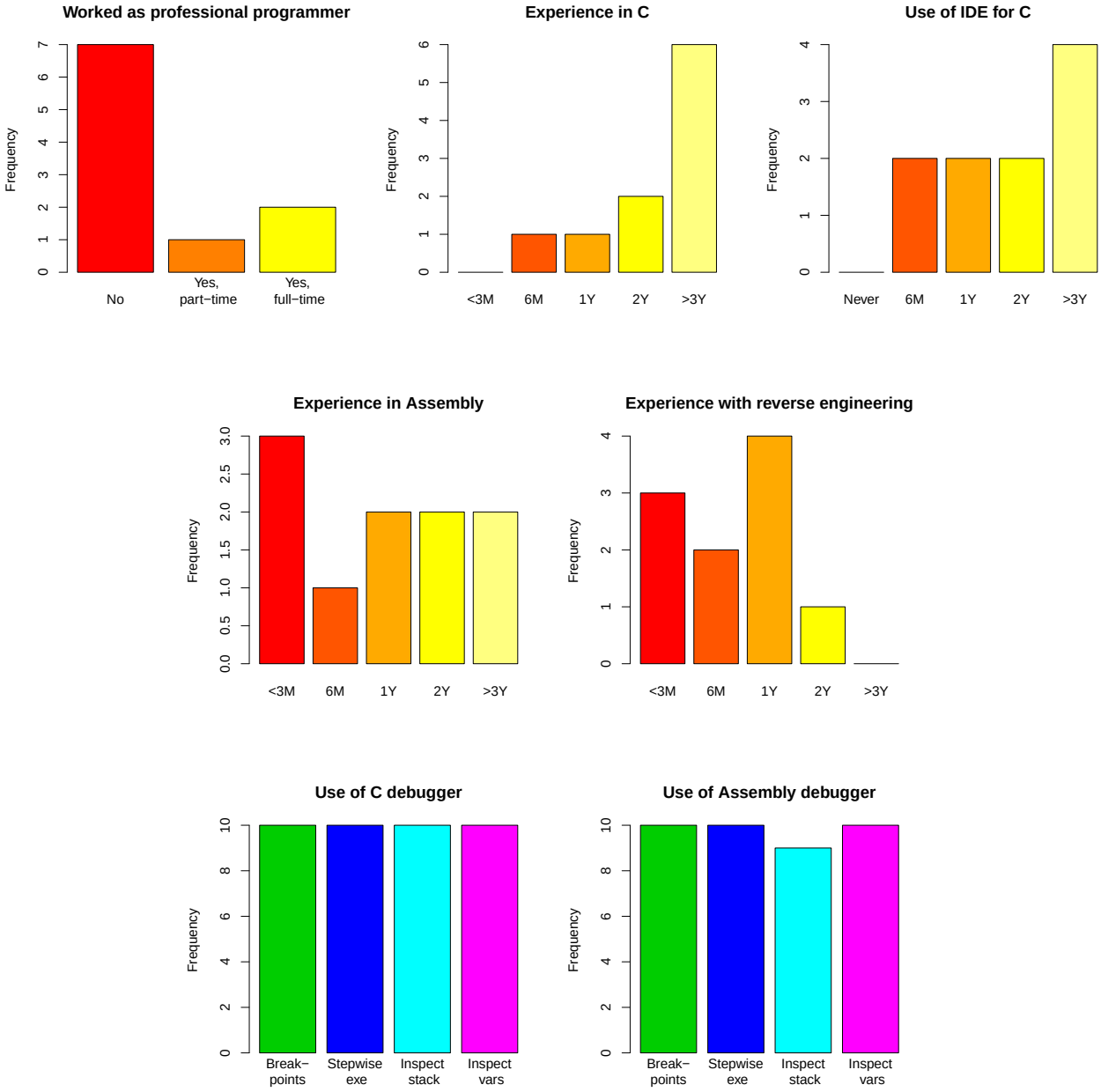
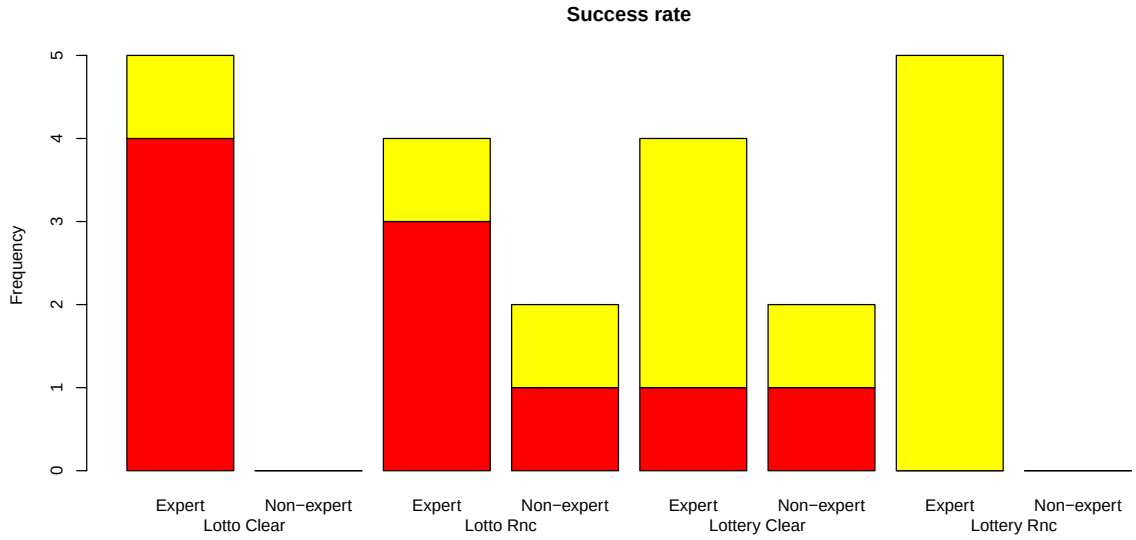


Figure 23: Demographics of UGent's subjects

Percentage success rate			
Lotto clear	Lotto RNC	Lottery clear	Lottery RNC
80%	60%	20%	0%
Average attack time (minutes)			
Lotto clear	Lotto RNC	Lottery clear	Lottery RNC
23.5	**141.0**	111	NA
Correlation between experience and success rate			
Professional prog	Exp in C	Exp in Assembly	Exp with reveng
0.02	0.10	-0.04	0.26

Figure 24: UGent experiment: effectiveness of RNC protection (boldface values have statistical significance at level 0.05)

cess rate, depicted as two stacked histograms for each pair program-treatment. The lower (red) histogram depicts the number of successful attacks for the given program-treatment; the upper (yellow) histogram depicts the number of unsuccessful attacks.
When moving from the clear version of the program to the RNC-obfuscated one, the success rate decreases substantially. No participant was able to successfully attack program Lottery obfuscated with RNC in this experiment. The percentage success rate is shown in the table at the bottom of Figure 24 (upper part of the table). The probability of a successful attack decreases by 20% both in program Lotto and Lottery when the obfuscated version is provided instead of the clear version.
In Figure 25 (upper part) success rate is split by subjects' experience. We classify a subject as *Expert* when she/he has one year or more of experience with C. No clear pattern emerges from the plot. ANOVA table (reported in Figure 25, lower part) confirms no statistical significant effect of experience.
In Figure 24, boxplots on the top-right show the distribution of the attack time across subjects, for Lotto-clear and for Lotto-RNC. Since no subject successfully attacked Lottery-RNC, only boxplots for Lotto are shown. When moving from the clear to the obfuscated version of Lotto, the attack time increases substantially. The average attack time in minutes is reported in the table at the bottom of Figure 24 (middle part of the table). According to the Wilcoxon non parametric test, the difference between the attack time observed for Lotto-clear and the attack time observed for Lotto-RNC is statistically significant at level 0.05. For this reason the average attack time for Lotto is shown in boldface in the table at the bottom of Figure 24. On average, the attack time on obfuscated code is six times higher than the attack time on clear code.
We have investigated the relationship between the subjects' experience and their capability to successfully complete the attack task. Results of the Spearman's test for statistical correlation are shown in the table at the bottom of Figure 24 (bottom part). Subjects' experience has been measured as the answer to the pre-questionnaire questions about their previous activities as pro-



	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Application	1	1.80	1.80	10.96	**0.0052**
Treatment	1	0.20	0.20	1.22	0.2885
Experience	1	0.36	0.36	2.16	0.1634
Application:Treatment	1	0.04	0.04	0.27	0.6111
Application:Experience	1	0.10	0.10	0.61	0.4483
Residuals	14	2.30	0.16		

Figure 25: Ugent experiment: effectiveness of RNC protection split by subjects experience (boldface values have statistical significance at level 0.05)

Factor	***p*-value**
Program	**0.00549**
Protection	0.30091
Program:Protection	1.00000

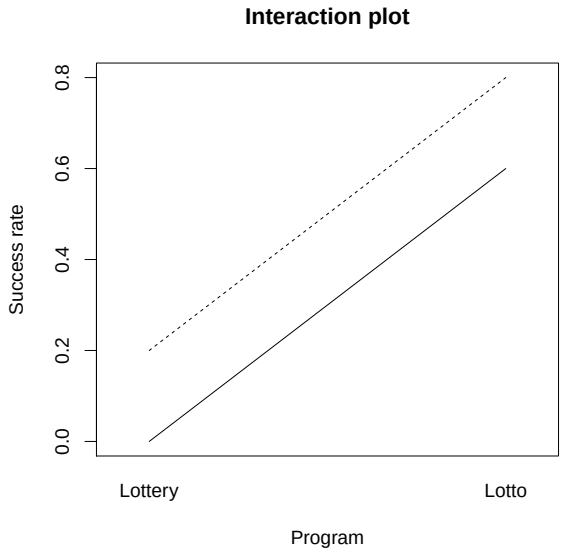


Figure 26: UGent experiment: interaction of *Program* (Lotto vs. Lottery) and *Protection* (Clear vs. RNC) with *Success rate* (boldface values have statistical significance at level 0.05; a dashed line indicates the clear version; a solid line the obfuscated version)

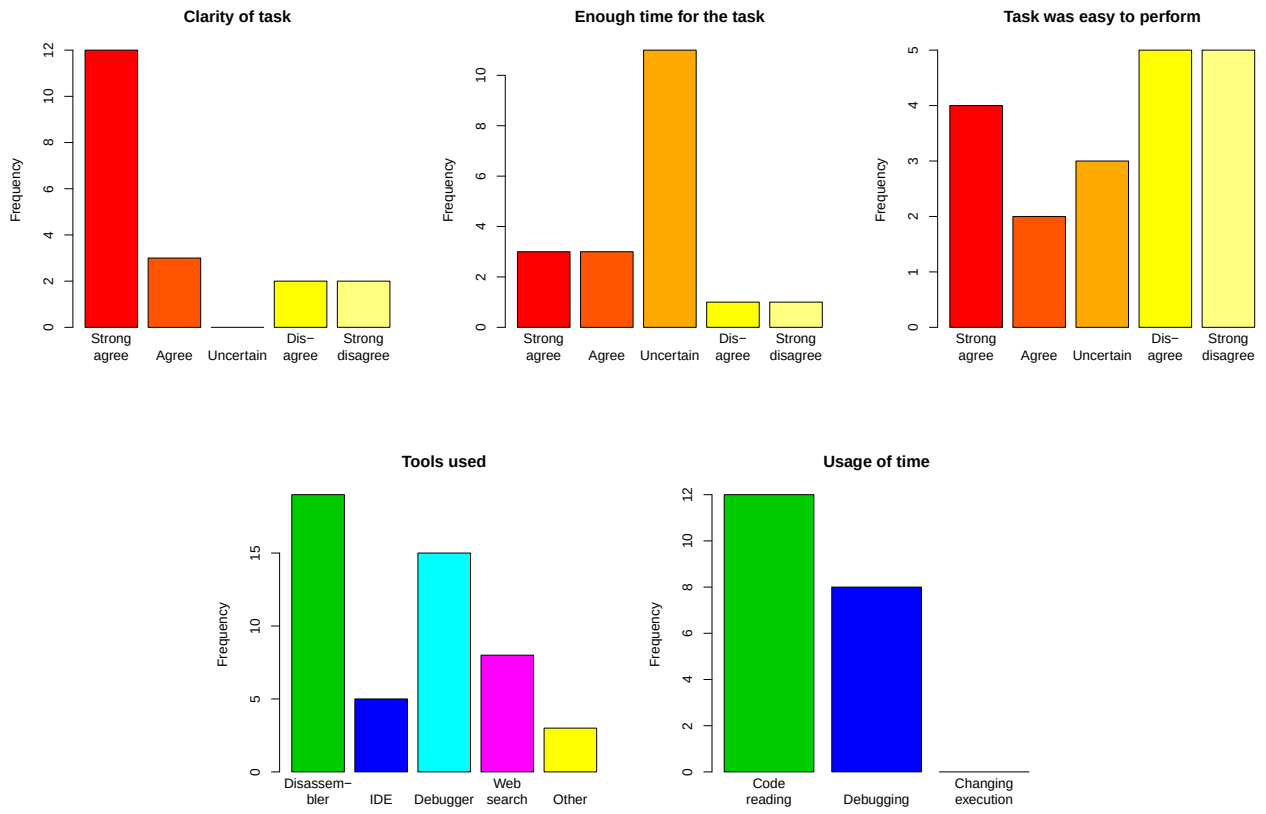


Figure 27: Post-questions answered by UGent's subjects

fessional programmers, their knowledge of C and Assembly, and their previous experience with binary reverse engineering. None of these measures of experience has a strong correlation with the subjects' success rate. In particular, no correlation achieves statistical significance at level 0.05. There is however a weak positive correlation between experience in C and in reverse engineering, and the subjects' capability to perform a successful attack.

Figure 26 shows the results of the interaction analysis. The outcome of the two-way ANOVA test is shown on the left. The specific program being considered (either Lotto or Lottery) affects significantly the success rate, with Lotto being substantially easier to attack than Lottery, as apparent from the interaction plot depicted on the right. The protection does not have a statistically significant effect (at level 0.05) on the success rate, as indicated also at the bottom in Figure 24 (upper part of the table), where the percentage success rates are not in boldface. This might be due to the small number of data points available from the experiment. Moreover, there is no significant interaction between the two factors *Program* and *Protection*, considered jointly, and the success rate. This is also visible graphically in the interaction plot displayed on the right, where the dashed line (for the clear code) and the solid line (for the obfuscated code) are parallel.

In summary, based on the data collected in UGent's experiment, we can answer research question RQ1 as follows:

> ***RQ1**: when attackers work on the binary code, RNC data obfuscation reduces the probability of a successful attack by 20% and increases the attack time by 6 times. The actual effectiveness of the protection is strongly dependent on the program being protected. Subjects with previous experience in C and in binary reverse engineering seem to have slightly more chances of completing the attack task successfully.*

According to the post-questionnaire (see Figure 27), tasks were generally clear to subjects, but the

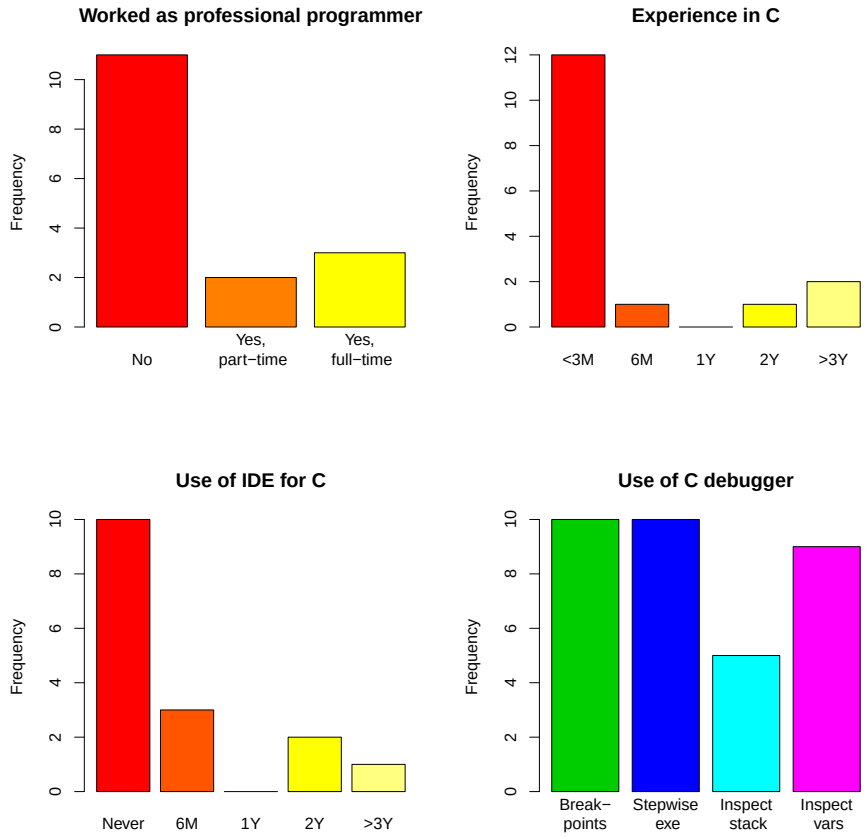
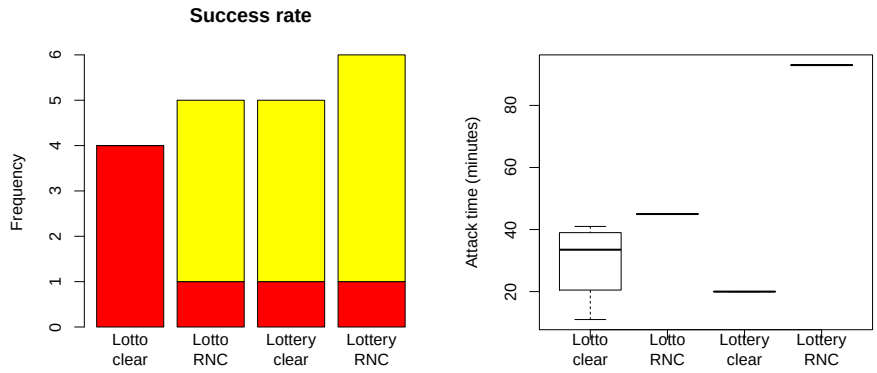


Figure 28: Demographics of FBK's subjects

time to perform them was judged neither sufficient nor insufficient (most subjects were uncertain about this question; see Figure 27). This might be due to the difficulty of the tasks to be carried out on obfuscated code. In fact, histograms on the top-right in Figure 27 indicate a mixed perception about the difficulty of tasks, with some prevalence of "difficult" over "easy".
UGent's subjects, who worked on binary code, used mostly the disassembler and the debugger. They also navigated the web to find solutions to the problems they encountered. They spent most time reading and debugging the program to be attacked.

6.6.2 FBK results

At FBK, subjects are MSC students attending the course of *Security Testing*. As such, they have at least a basic background on C its standard API. However, they are fluent on other programming languages (e.g., Java) that have been used to deliver course projects. Before the experiment, they attended and introduction lecture on code protection and code obfuscation in general.
Differently from the UGhent experiment, in this case the lab was limited to 2 hours and subjects were not supposed to continue on the assignment as homework after the lab.
Figure 28 shows some descriptive statistics about the participants involved in the FBK's experiment, which was carried out on C source code. These data were collected by means of the pre-experiment questionnaire reported in Appendix D.
In total, 16 MSc students participated to the experiment, however some of them attended only one of the two sessions. Most of them had no previous experience as professional programmers. The majority declared also limited (less than three months) experience with the C programming language and with a C programming IDE. All subjects declared good knowledge of the features



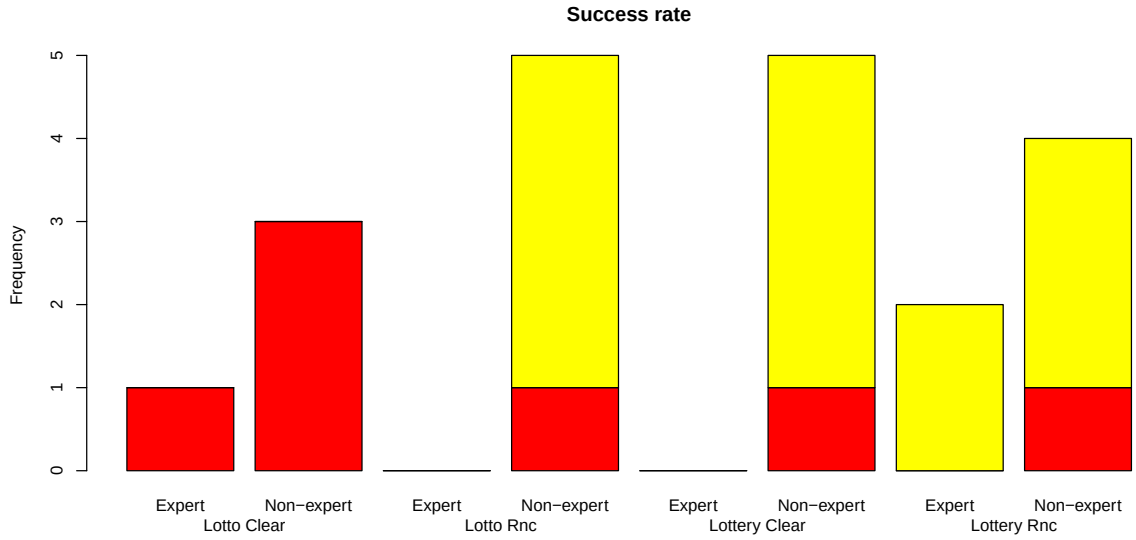
Percentage success rate			
Lotto clear	Lotto RNC	Lottery clear	Lottery RNC
100%	**20%**	20%	16%

Average attack time (minutes)			
Lotto clear	Lotto RNC	Lottery clear	Lottery RNC
29.7	**45.0**	20	93

Correlation between experience and success rate	
Professional programmer	Experience in C
-0.22	0.14

Figure 29: FBK experiment: effectiveness of RNC protection (boldface values have statistical significance at level 0.05)

of C debuggers, with the exception of call stack inspection, which was known only to half of the respondents.
Overall, subjects participating in this experiment have little professional experience and limited experience with C; they know the C debugger reasonably well.
Figure 29 shows the results of FBK's experiment. Histograms on the top-left show the success rate, depicted as two stacked histograms for each pair program-treatment. The lower (red) histogram depicts the number of successful attacks for the given program-treatment; the upper (yellow) histogram depicts the number of unsuccessful attacks.
When moving from the clear version of the program to the RNC-obfuscated one, the success rate decreases substantially. The percentage success rate is shown in the table at the bottom of Figure 29 (upper part of the table). The probability of a successful attack decreases by 80% in program Lotto and by 4% in the case of Lottery when the obfuscated version is provided instead of the clear version. In the case of Lotto, such a difference is statistically significant at level 0.05. On average, across programs we observe a 42% decrease of the probability of a successful attack mounted against the RNC protected version of the program. However, it should be noticed that the actual decrease changes substantially between the two considered programs, Lotto-clear being much easier to attack than Lottery-clear. Such a difference tends to disappear on the RNC-obfuscated versions. Hence, obfuscated programs seem to be equally difficult to attack, regardless of the difficulty of attack for the original, clear programs.
In Figure 30 (upper part) success rate is split by subjects' experience. We classify a subject as *Expert* when she/he has one year or more of experience with C. No clear pattern emerges from the plot. ANOVA table (reported in Figure 30, lower part) confirms no statistical significant effect of experience.
In Figure 29, boxplots on the top-right show the distribution of the attack time across subjects, for Lotto-clear/RNC and for Lottery-clear/RNC. When moving from the clear to the obfuscated



	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Application	1	0.69	0.69	4.12	0.0619
Treatment	1	0.71	0.71	4.20	0.0596
Experience	1	0.02	0.02	0.11	0.7457
Application:Treatment	1	0.75	0.75	4.50	0.0523
Application:Experience	1	0.03	0.03	0.18	0.6789
Residuals	14	2.35	0.17		

Figure 30: FBK experiment: effectiveness of RNC protection split by subjects experience (boldface values have statistical significance at level 0.05)

Factor	**p-value**
Program	**0.0488**
Protection	**0.0468**
Program:Protection	**0.0449**

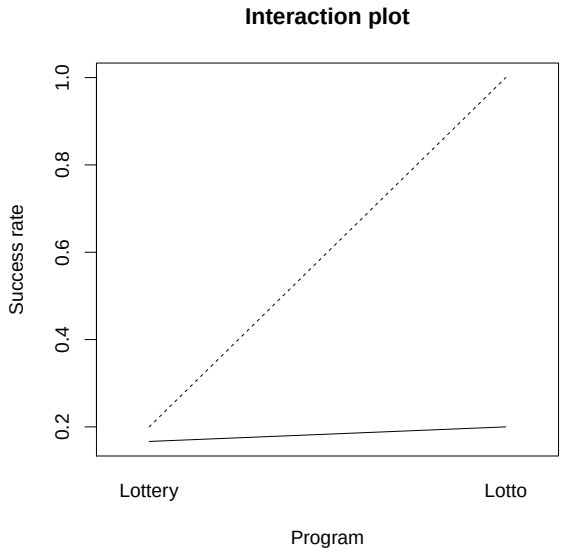


Figure 31: FBK experiment: interaction of *Program* (Lotto vs. Lottery) and *Protection* (Clear vs. RNC) with *Success rate* (boldface values have statistical significance at level 0.05; a dashed line indicates the clear version; a solid line the obfuscated version)

version of Lotto/Lottery, the attack time increases substantially. The average attack time in minutes is reported in the table at the bottom of Figure 29 (middle part of the table). According to the Wilcoxon non parametric test, the difference between the attack time observed for Lotto-clear and the attack time observed for Lotto-RNC (shown in boldface) is statistically significant at level 0.05. Even if a remarkable difference in the amount of time required to attack the clear and the obfuscated version of Lottery (20 versus 93 minutes), due to the small number of successful cases on Lottery, this observation is not statistical significant.
For Lotto the attack time on RNC-obfuscated code is 1.5 times higher than on clear code; for Lottery it is 4.6 times higher. On average, the attack time on obfuscated code is 3 times higher than the attack time on clear code.
We have investigated the relationship between the subjects' experience and their capability to successfully complete the attack task. Results of the Spearman's test for statistical correlation are shown in the table at the bottom of Figure 29 (bottom part). Subjects' experience has been measured as the answer to the pre-questionnaire questions about their previous activities as professional programmers and their knowledge of C. None of these measures of experience has a strong correlation with the subjects' success rate. In particular, no correlation achieves statistical significance at level 0.05. There is however a weak positive correlation between experience in C and the subjects' capability to perform a successful attack. Quite surprisingly, there is also a weakly negative correlation between previous experience as professional programmers and attack success rate. This might indicate that the typical industrial programming tasks are quite different from the tasks carried out to attack a program. Then again, that difference would also exist between typical university programming tasks and the attack task. Several project partners discussed this correlation, and decided that it is impossible to confidently pinpoint the reason for this correlation. It is therefore purely accidental, as a result of the low sample size.
Figure 31 shows the results of the interaction analysis. The outcome of the two-way ANOVA test is shown on the left. The specific program being considered (either Lotto or Lottery) affects significantly the success rate, with Lotto-clear being substantially easier to attack than Lottery-clear, as apparent from the interaction plot depicted on the right (dashed line). The presence of a protection has a statistically significant effect on the success rate, as indicated also at the bottom in Figure 29 (upper part of the table), where the percentage success rates are in boldface for Lotto. Protected programs are substantially more difficult to attack than clear code programs. Moreover, there is a statistically significant interaction between the two factors *Program* and *Protection*, considered jointly, and the success rate. This is also visible graphically in the interaction plot displayed on the right, where the dashed line (for the clear code) and the solid line (for the obfuscated code) diverge. On programs that are easier to attack (Lotto) the application of data obfuscation increases the level of protection much more than for programs (Lottery) that are difficult to attack even when distributed as clear code. In other words, the effect of a protection gets amplified if it is applied to programs that are otherwise relatively easy to attack.
In summary, based on the data collected in FBK's experiment, we can answer research question RQ1 as follows:

> ***RQ1**: when attackers work on the C source code, RNC data obfuscation reduces the probability of a successful attack by 42% and increases the attack time by 3 times. The actual effectiveness of the protection is strongly dependent on the program being protected. Moreover, the magnitude of the protection is greater when the program to be protected is relatively easy to attack. Subjects with previous experience in C seem to have slightly more chances of completing the attack task successfully.*

According to the post-questionnaire (see Figure 32), tasks were generally clear to subjects, but the time to perform them was not always judged to be sufficient (many subjects were uncertain about this question; see Figure 32). This might be due to the difficulty of the tasks to be carried out on obfuscated code. In fact, histograms on the top-right in Figure 32 have a peak on "Disagree", indicating the prevalence of judgment "task was difficult to perform" over "task was easy

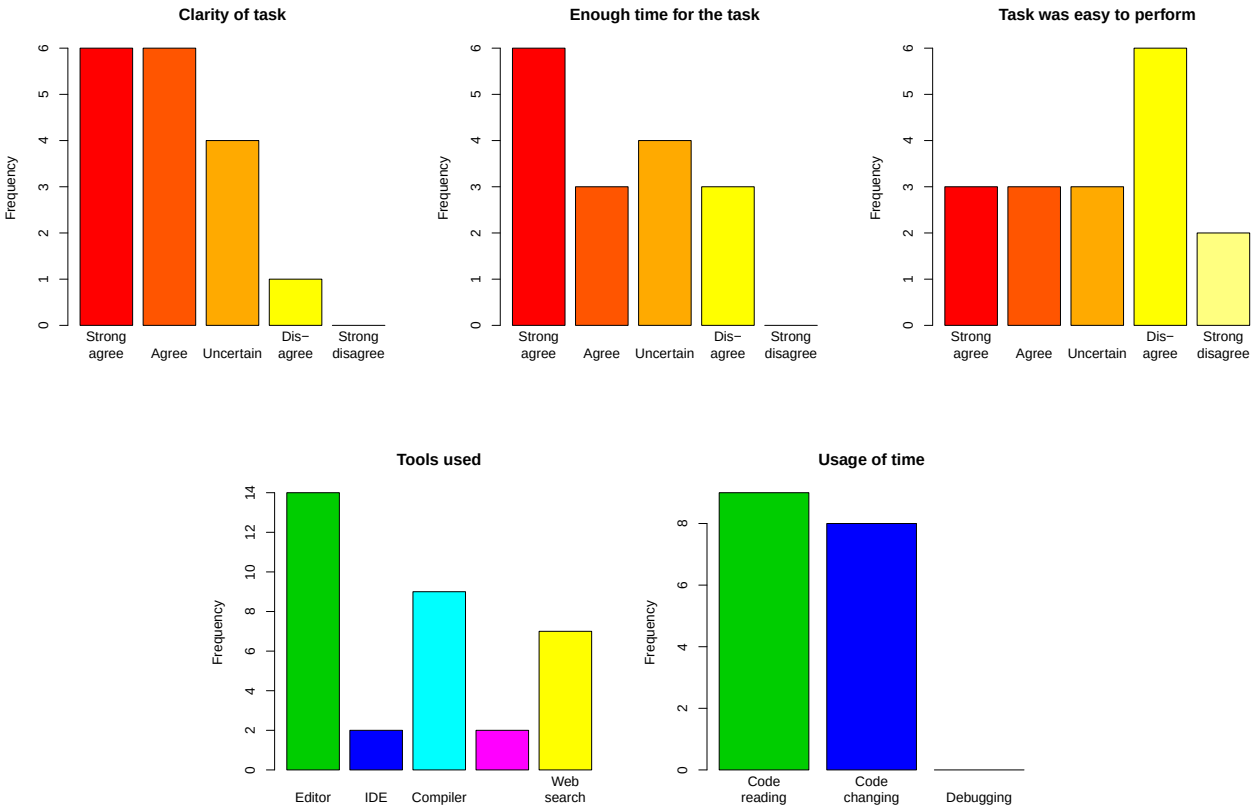


Figure 32: Post-questions answered by FBK's subjects

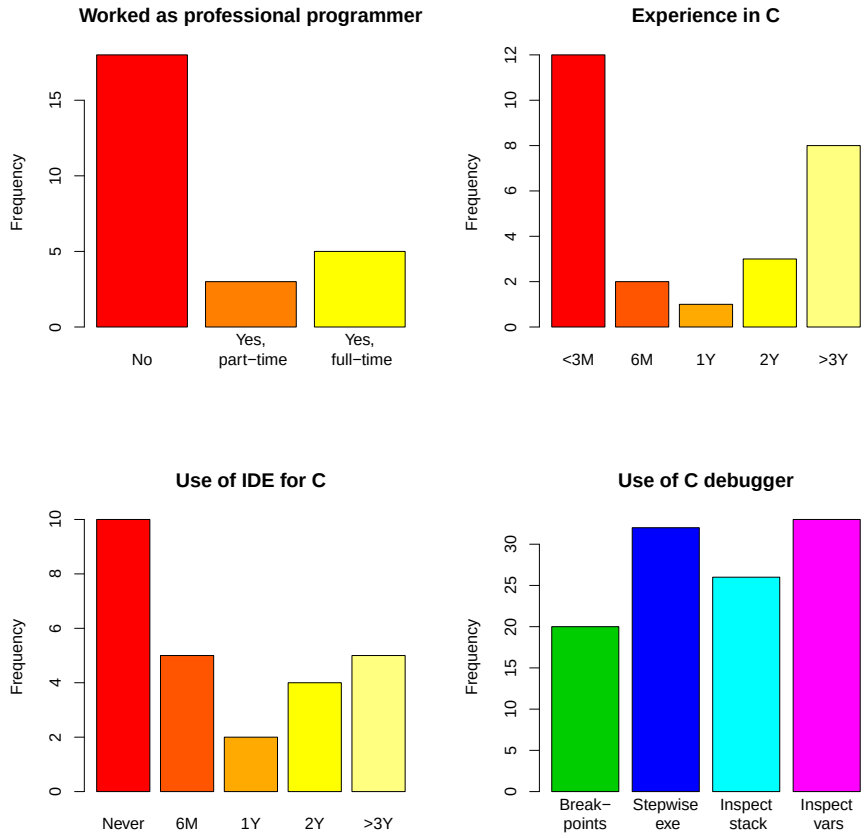
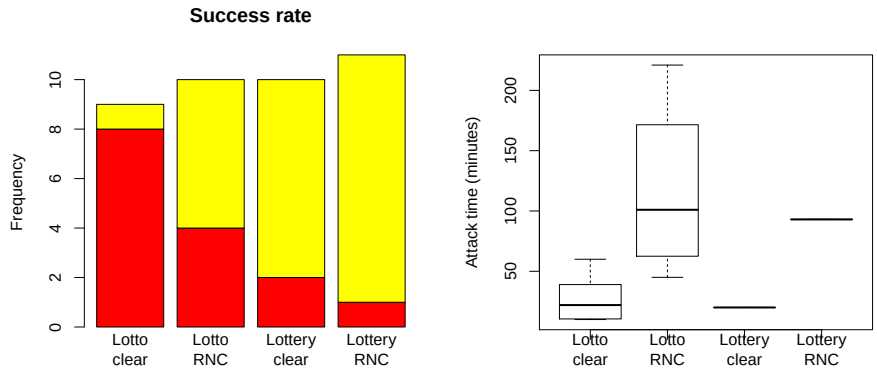


Figure 33: Demographics of all participating subjects

to perform".
FBK's subjects, who worked on C source code, used mostly the editor and the compiler. They also navigated the web to find solutions to the problems they encountered. They spent most time reading and changing the program to be attacked.

6.6.3 Overall results

Figure 33 shows some descriptive statistics about all participants involved in the ASPIRE experiments. These data were collected by means of the pre-experiment questionnaire reported in Appendixes C, D.
In total, 26 MSc students participated to the two sessions of the data obfuscation experiments. Most of them had no previous experience as professional programmers. Subjects are split into two groups, having respectively limited (less than three months) and substantial (more than three years) experience with the C programming language and with a C programming IDE. These two groups correspond roughly to UGent's and FBK's subjects, respectively. All subjects declared good knowledge of the features of C debuggers.
Figure 34 shows the overall results of ASPIRE's data obfuscation experiments. Histograms on the top-left show the success rate, depicted as two stacked histograms for each pair program-treatment. The lower (red) histogram depicts the number of successful attacks for the given program-treatment; the upper (yellow) histogram depicts the number of unsuccessful attacks.
When moving from the clear version of the program to the RNC-obfuscated one, the success rate decreases substantially. The percentage success rate is shown in the table at the bottom of Figure 34 (upper part of the table). The probability of a successful attack decreases by 48% in program Lotto



Percentage success rate			
Lotto clear	Lotto RNC	Lottery clear	Lottery RNC
88%	**40%**	20%	9%

Average attack time (minutes)			
Lotto clear	Lotto RNC	Lottery clear	Lottery RNC
26.6	**117.0**	20	93

Correlation between experience and success rate	
Professional programmer	Experience in C
-0.10	0.10

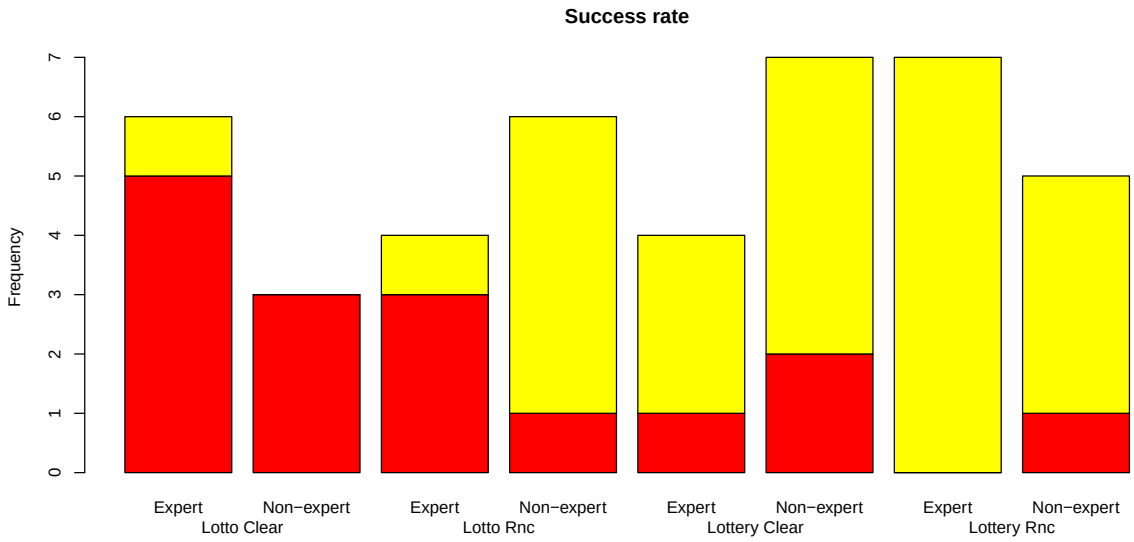
Figure 34: Overall results: effectiveness of RNC protection (boldface values have statistical significance at level 0.05)

and by 11% in the case of Lottery when the obfuscated version is provided instead of the clear version (the difference is statistically significant at level 0.05 in the case of Lotto). On average, across programs we observe a 30% decrease of the probability of a successful attack mounted against the RNC protected version of the program. However, it should be noticed that the actual decrease changes substantially between the two considered programs, probably because Lottery-clear is already quite difficult to attack as compared to Lotto-clear.

In Figure 35 (upper part) success rate is split by subjects' experience. We classify a subject as *Expert* when she/he has one year or more of experience with C. As we can see from the graph, when working on Lotto expert subjects have higher success rate than non-expert subjects both on clear code and on code obfuscated with RNC. However, an opposite trend is observed on Lottery where expert subjects have lower success rate than non-expert subjects (both on clear and on obfuscated code). This fact trend is evident also in the ANOVA table (reported in Figure 35, lower part) where experience is not an influencing effect by itself, but only considered together with the main treatment and the application.

In Figure 34, boxplots on the top-right show the distribution of the attack time across subjects, for Lotto-clear/RNC and for Lottery-clear/RNC. When moving from the clear to the obfuscated version of Lotto/Lottery, the attack time increases substantially. The average attack time in minutes is reported in the table at the bottom of Figure 34 (middle part of the table). According to the Wilcoxon non parametric test, the difference between the attack time observed for Lotto-clear and the attack time observed for Lotto-RNC (shown in boldface) is statistically significant at level 0.05. For Lotto the attack time on RNC-obfuscated code is 4.3 times higher than on clear code; for Lottery it is 4.6 times higher. On average, the attack time on obfuscated code is 4.4 times higher than the attack time on clear code.

We have investigated the relationship between the subjects' experience and their capability to successfully complete the attack task. Results of the Spearman's test for statistical correlation



	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Application	1	2.38	2.38	16.05	**0.0003**
Treatment	1	0.84	0.84	5.63	**0.0238**
Experience	1	0.12	0.12	0.83	0.3704
Application:Treatment	1	0.28	0.28	1.88	0.1801
Application:Experience	1	0.25	0.25	1.72	0.1997
Treatment:Experience	1	0.07	0.07	0.48	0.4942
Application:Treatment:Experience	1	0.68	0.68	4.58	**0.0401**
Residuals	32	4.75	0.15		

Figure 35: Overall results: effectiveness of RNC protection split by subjects experience (boldface values have statistical significance at level 0.05)

Factor	***p*-value**
Program	**0.000471**
Protection	**0.028770**
Program:Protection	0.144256

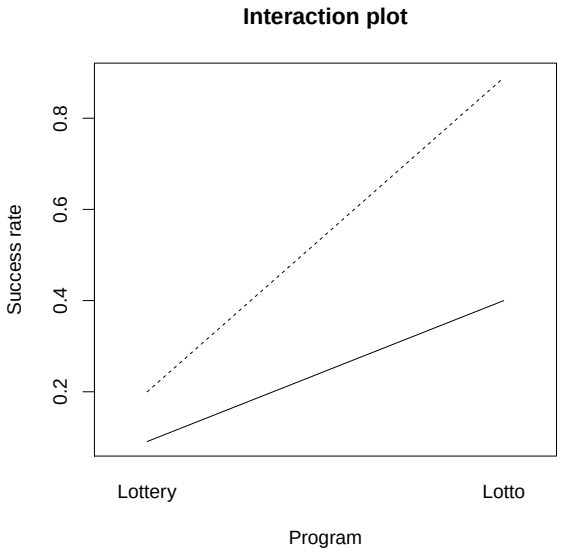


Figure 36: Overall results: interaction of *Program* (Lotto vs. Lottery) and *Protection* (Clear vs. RNC) with *Success rate* (boldface values have statistical significance at level 0.05; a dashed line indicates the clear version; a solid line the obfuscated version)

Figure 37: Post-questions answered by all participating subjects

are shown in the table at the bottom of Figure 34 (bottom part). Subjects' experience has been measured as the answer to the pre-questionnaire questions about their previous activities as professional programmers and their knowledge of C. None of these measures of experience has a strong correlation with the subjects' success rate. In particular, no correlation achieves statistical significance at level 0.05. There is however a weak positive correlation between experience in C and the subjects' capability to perform a successful attack. Quite surprisingly, there is also a weakly negative correlation between previous experience as professional programmers and attack success rate. This might indicate that the typical industrial programming tasks are quite different from the tasks carried out to attack a program.
Figure 36 shows the results of the interaction analysis. The outcome of the two-way ANOVA test is shown on the left. The specific program being considered (either Lotto or Lottery) affects significantly the success rate, with Lotto being substantially easier to attack than Lottery, as apparent from the interaction plot depicted on the right. The protection also has a statistically significant effect (at level 0.05) on the success rate, as indicated also at the bottom in Figure 34 (upper part of the table), where the percentage success rates for Lotto are in boldface. Overall, there is no significant interaction between the two factors *Program* and *Protection*, considered jointly, and the success rate. This is also visible graphically in the interaction plot displayed on the right, where the dashed line (for the clear code) and the solid line (for the obfuscated code) are almost parallel (actually, there is some divergence, indicating that the effect of protection is higher on Lotto than on Lottery, but such divergence does not reach statistical significance).
In summary, based on the data collected in all data obfuscation experiment, we can answer research question RQ1 as follows:

> ***RQ1**: when attackers work either on the C source code or on the binary code, RNC data obfuscation reduces the probability of a successful attack by 30% and increases the attack time by 4.4 times. The actual effectiveness of the protection is strongly dependent on the program being protected. Subjects with previous experience in C and no previous experience as professional programmers seem to have slightly more chances of completing the attack task successfully.*

According to the post-questionnaire (see Figure 37, where only questions in common between binary code and C source code assignments are considered), tasks were generally clear to all participating subjects, but the time to perform them was not always judged to be sufficient (many subjects were uncertain about this question; see Figure 37). This might be due to the difficulty of the tasks to be carried out on obfuscated code. In fact, histograms on the top-right in Figure 37 are slightly peaked around "Disagree", indicating some prevalence of judgment "task was difficult to perform" over "task was easy to perform".

6.7 Comparison Results Source Code Attacks vs. Binary Code Attacks

The results of the attack experiments conducted at UGent on binary code and at FBK on source code are mostly in line with each other. This was not a big surprise to the researchers.
The investigated RNC data obfuscation is performed entirely on an application's source code. As the program was compiled at UGent without compiler optimizations enabled, the attack subjects there studied disassembled code that was mostly a straightforward translation of the original source code to the assembly level.
After applying the obfuscation, the increase in source code complexity as perceived by a programmer experienced with C is not significantly different from the increase in assembler code complexity as perceived by a binary code reverse engineer. Moreover, the one aspect that is very different between source code and binary code attacks, was omitted from the experiment at UGent: the actual editing of the code to alter its behavior. In source code, the required changes, once identified, are trivial to implement. They hence did not really contribute to the attack complexity in FBK. And at UGent, they were omitted. So in both experiments, the attack complexity was more or less bound by the reverse engineering part, not by the tampering part of the attack.
This experiment therefore to some extent validates the idea of the project's PIs that for some forms of obfuscations, performing experiments in source code can be a good proxy for experiments at the binary level.

6.8 Threats to validity

The main threats to the validity of this experiment belong to the conclusion, internal, construct and external validity threat categories.
Conclusion validity threats concern the relationship between treatment and outcome. We use statistical tests to draw our conclusions. Inability to reject the null hypothesis exposes us to type II errors (incorrectly accepting a false null hypothesis). We mitigate this threat by replicating the experiment four times, so as to increase the number of participants, N. In fact, the probability of a type II error can be reduced by increasing the sample size. We used two-way ANOVA for the interaction analysis. Although ANOVA requires data to be normally distributed and our data may not satisfy such assumption, when used for interaction analysis, two-way ANOVA is known to be quite robust with respect to deviations from normality [2].
Subjects are provided with the source code, not the binary code, because students are not proficient enough in binary and assembly code analysis at FBK, UEL and POLITO. This might make our conclusions not applicable when only the binary is available to attackers. To mitigate this threat, the replication at UGent is conducted on the binary.
Internal validity threats concern external factors that may affect the independent variable. Subjects are not aware of the experimental hypotheses. At FBK, subjects are not rewarded for the participation in the experiment. At UGent, students got a 50 euro gift certificate for their participation, but students that were identified as having insufficient experience with x86 binary code (i.e., no more experience than what they learn in UGent's courses on basic computer architecture and operating systems) were not allowed to participate. This ensured that only students with at least some passion for binary code and reverse engineering participated. All students were informed that they are not evaluated on their performance in doing the experiment.
Construct validity threats concern the relationship between theory and observation. They are mainly due to how we measure the effectiveness of data obfuscation. We manually assess the successful completion of each task in order to measure SR. During the labs, the experimenters made sure that times are accurately marked in the time sheets upon start and completion of the attack task.
External validity concerns the generalisation of the findings. In our experiment we considered two small programs, Lotto and Lottery, to allow for potentially successful attacks within experimental sessions with a limited time bound (2h). Results may not generalise to different programs. On the other hand, we expect that larger programs will be more difficult, not easier, to attack, both

with and without protections. Moreover, the chosen programs have been written by third parties, completely unrelated with ASPIRE, so as to ensure that they are not crafted to provide optimal application conditions for the ASPIRE protections. Further replications of the study on additional objects will corroborate the external validity of our findings.

6.9 Lessons Learned

The results of this first round of experiments have been discussed internally to the consortium. Moreover, we asked the ASPIRE Scientific Advisory Board to comment on them and to formulate potential recommendations for improvement. Here we summarize the points that we intend to change on the experimental design before the next replications:

- **Profiling questionnaire:** We will add new questions, intended to ask participants explicitly about their experience with C/C++ in projects, to estimate the projects' size (such as lines of code) to have a more objective quantification of their experience.

  Moreover, will add a new question on experience with Java. In fact, most of the MSC students are quire fluent on Java, acquired when working on many academic or open source projects. This familiarity with programming could influence the success rate of attack tasks, even if on a different, but quite similar, language.

- **Decompiled code:** The SAB suggested to use decompiled source code instead of original source code. We plan to check the output of a decompiler on code compiled with different levels of optimization, and verify if the result can be still used in the tasks. In fact, decompiled code might look strange at the first sight and very different than source code. In this case, the ability to understand decompiled code could be largely affected by the experience of subjects in working with decompiled code.

- **New obfuscations:** The SAB suggested not to limit to state-of-the-art protections, but to experiment also with new protections developed by the project. Even if this was already in our initial plan (code splitting in as new protection, and all the new protections will be experimented with professional hackers) we intend to go further and experiment also the new version of data obfuscation elaborated by the project. The second replication in UGhent was meant to compare clear code and code protected with Code Splitting. We are considering to change the plan and study the additional level of protection offered by the dynamic variant of XOR Masking (elaborated in task T2.1) with the state-of-the-art version of XOR Masking.

- **Training:** As suggested by the SAB, we plan to include a mandatory training phase before the experimental lab. The aim is to make sure that all the subjects are familiar with an attack task before the actual lab, so they do not waste time during the lab to familiarize with the experimental setting. This will help in limiting learning effect and to measure only the time actually spent in attacking the code.

6.10 Dates

The next two replications, with treatments T2 (XOR) and T3 (Var merge), of the data obfuscation experiment will be conducted at University of East London and at Politecnico of Torino in Autumn, 2015.

7 Code splitting experiment

Table 7 provides a schematic overview of the code splitting experiment. The *goal* of this experiment is to analyse the degree of protection offered by the ASPIRE code splitting technique, when it is applied to increasingly large code portions. Splitting larger code portions is expected to bring

Goal	Analyze the ability of code splitting to prevent malicious attacks and measure the performance penalty to be paid
Treatments	T0 = original code; T1 = small code portion split; T2 = medium code portion split; T3 = large code portion split
RQ1	How much does code splitting increase the attack time as compared to the attack time for the clear code?
RQ2	How do different code splitting choices (T1/2/3) affect the attack time?
RQ3	What is the performance impact of different code splitting (T1/2/3)?
Subjects	Students from FBK, UEL, POLITO and UGent (working on the binary code)
Objects	P1: space game
Tasks	Make spacecraft move faster by doubling the effect of a move
Metrics	Success rate; time to mount a successful attack
Design	Lab1: P1-T0; P1-T1/2/3 (Repl 1/3/4)

Table 7: Code splitting experiment

increased protection but also to cause increasing performance degradation, due to the need for more frequent synchronisations between the code remaining on the client and the code moved to the trusted server/device. In order to evaluate the impact of the split code size on both the level of protection and the performance penalty to be paid, we consider three treatments, in addition to the baseline T0, which is the original code: T1/T2/T3, associated with a small/medium/large code portion being moved from the client to the trusted server/device.
In each replication of the experiment, one of the three split code size (T1/T2/T3) is evaluated in comparison with the clear code (T0). This allows for a direct measurement of the increased attack effort associated with each level of protection and it also allows for a direct measurement of the associated performance degradation. Moreover, by comparing the data collected in the three replications where the split code portion is increasingly large (i.e., Repl1, Repl3, Repl4), it is possible to quantitatively assess the trade off between degree of protection and associated performance penalty. The replication at UGent (Repl2) will be carried out directly on the binary code, since students at UGent are specifically trained on binary and assembly code analysis.
The three replications Repl1, Repl3, Repl4 of this experiment with increasingly large split code will be conducted respectively at POLITO, FBK and UEL, and will evaluate the treatment (T1/T2/T3) in comparison with the baseline treatment T0 (clear code). The replication at UGent will evaluate one of the three treatments (not yet decided, but probably T2) against T0 when the binary code instead of the source code is provided to the students. The tentative dates for the four replications of this experiment are shown in Table 7.

7.1 Research questions

The experiment aims at answering the following three research questions:

- **RQ1**: How much does code splitting increase the attack time as compared to the attack time for the clear code?
- **RQ2**: How do different code splitting choices (T1/2/3) affect the attack time?
- **RQ3**: What is the performance impact of different code splitting (T1/2/3)?

The first research question is about the effectiveness of code splitting as a code protection technique. The evidence collected in the four replications of the experiment will be used to assess the

average increased attack time provided by code splitting, regardless of the split code size. The second research question aims at contrasting the degree of protection offered at different split code size. The third research question deals with the performance degradation associated with code splitting, depending on the size of the code portion being split.

7.2 Object

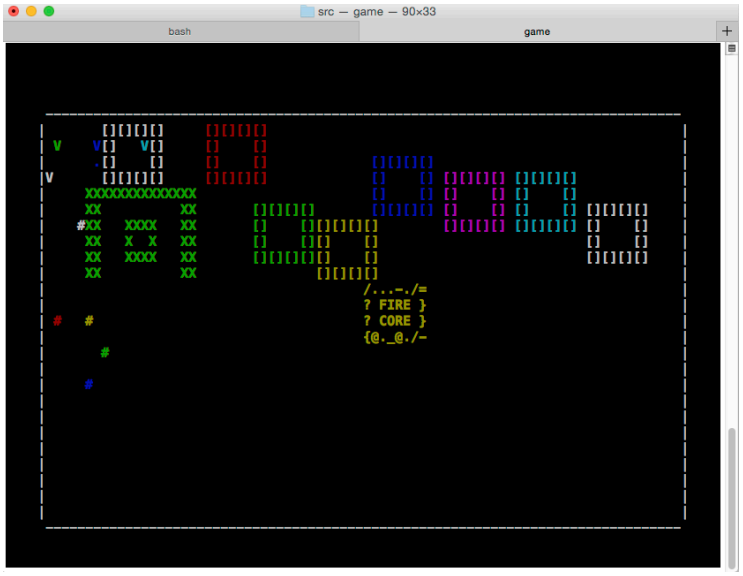


Figure 38: Screenshot of SpaceGame

The object of this experiment is an open source C program, SpaceGame, obtained from SourceForge. SpaceGame is a demonstrator for the framework GAME (Geometrical Ascii Multigame Environment), a C language framework for creating geometrical games using `ncurses` text screens. While GAME was originally created for unix platforms, it can be ported to more systems, because it uses standard ansi C. The framework and the demonstrator amount to 1,873 SLoC (measured by `sloccount`), including header files. Players can move on the screen by means of the numeric keyboard or by pressing the keys for characters 'h', 'j', 'k', 'l' ('s' stops the game, while 'q' quits it). Figure 38 shows a screenshot of SpaceGame.
The attack task to be executed by the experiment subjects on SpaceGame aims at gaining an unfair advantage over other competing players:

Attack task: Modify the source code of SpaceGame so as to move twice as fast as allowed by the game rules. Specifically, each key press must translate into a 2-character length move, instead of a 1-character move.

While in the clear code the required modification consists just of changing the unitary increment or decrement of the position into a double increment/decrement, when the code handling player movements is moved from client to server, the modification to be done becomes increasingly more difficult, depending on the split code size. It might for instance involve a double function call that replaces a single call, or a modification of some client-server messages.

7.3 Metrics

The metrics collected to answer research questions RQ1, RQ2 and RQ3 are:

AT: Attack time

SR: Success rate

ET: Execution time

Subjects are asked to mark down the start and end time when starting and after finishing the attack task, so one key metrics collected during the experiment is the attack time (AT). Subjects are also asked to send the attacked code to the experimenters, who manually verify if the attack was successful or not. Metrics AT is meaningful only for successful attacks. The proportion of successful attacks provides a second metrics, which complements AT, called success rate (SR). SR measures the proportion of subjects that successfully completed the attack task either on the original code or on code protected by code splitting.
The execution time (ET) measures the time for a complete execution of SpaceGame under a predefined interaction scenario. In order to obtain meaningful and comparable execution times, a program driver is used to stimulate the program without requiring any user intervention. The driver sends a predefined key sequence to SpaceGame, so as to simulate the interaction of the user with the game. To accommodate for random fluctuations in the execution time measurements, ET is measured multiple (100) times.
The performance overhead (PO) is the relative increase of the average execution time between split code and original code:

$$PO = \frac{\overline{ET(P')} - \overline{ET(P)}}{\overline{ET(P)}} \tag{19}$$

where P, P' indicate the original and protected program, respectively
Based on the metrics chosen to quantify the effectiveness and penalty of the code splitting protection, we can formulate null and alternative hypotheses associated with research questions RQ1, RQ2 and RQ3:

- **H1-AT**$_0$: There is no difference in AT between subjects working on split and subjects working on original code
- **H1-SR**$_0$: There is no difference in SR between subjects working on split and subjects working on original code
- **H2-AT**$_0$: There is no difference in AT between subjects working on code with a small/medium/large code portion being split
- **H2-SR**$_0$: There is no difference in SR between subjects working on code with a small/medium/large code portion being split
- **H3-ET**$_0$: There is no difference in performance between original program and split program (i.e., PO ~ 0)
- **H1-AT**$_a$: Code splitting increases AT for subjects working on split code as compared to subjects working on original code
- **H1-SR**$_a$: Code splitting decreases SR for subjects working on split code as compared to subjects working on original code
- **H2-AT**$_a$: The code split size affects AT
- **H2-SR**$_a$: The code split size affects SR
- **H3-ET**$_a$: Code splitting introduces a non negligible performance overhead (i.e., PO > 0)

In addition to the metrics AT, SR and ET, we ask subjects to answer a pre-questionnaire and a post-questionnaire. The pre-questionnaire collects information about the abilities and experience of the involved subjects. This is very important to analyse the effect of ability and experience in the successful completion of the attack tasks either on original or on split code. The post-questionnaire collects information about clarity and difficulty of the task, availability of sufficient time to complete it, and on the tools used and the activities carried out to complete the task.

7.4 Design

The design of the family of experiments associated with replications Repl1, Repl3, Repl4 aims at exploring the trade off between increased protection and performance penalty associated with different levels of code splitting. Specifically, three levels of code splitting are applied in these three replications, T1/T2/T3 (with respectively a small/medium/large code portion being split), while T0 indicates no code splitting at all.

Lab	**P1-T0**	**P1-T'**
Lab1	G1	G2

Table 8: Design for the code splitting experiment: group G1 is assigned object P1 in its original form, while group G2 is assigned P1 protected with code splitting T' (either of T1/T2/T3), in a single lab (Lab1)

Table 8 shows the design of each experimental session (Lab1). Subjects are divided into two groups, G1 and G2. Object P1 (SpaceGame) is provided as clear code (T0) or split code code (T'). In the latter case, the amount of code splitting varies across replications (T' = T1, T2 or T3 depending on the replication).

7.5 Statistical analysis

The difference between the output variable (AT, SR, ET) obtained under different treatments (original code vs. split code) is tested using the Wilcoxon non-parametric statistical test, assuming significance at a 95% confidence level (α=0.05); so we reject the null-hypotheses having p-value$<$0.05. Regarding the analysis of pre and post questionnaires whose answers are on a Likert scale, we test the difference of the medians by means of the non-parametric Mann-Whitney statistical test, assuming significance at a 95% confidence level (α=0.05).

7.6 Threats to validity

The main threats to the validity of this experiment belong to the conclusion, internal, construct and external validity threat categories.
Conclusion validity threats concern the relationship between treatment and outcome. We use statistical tests to draw our conclusions. Inability to reject the null hypothesis exposes us to type II errors (incorrectly accepting a false null hypothesis). We mitigate this threat by replicating the experiment four times, so as to increase the number of participants, N. In fact, the probability of a type II error can be reduced by increasing the sample size. Subjects are provided with the source code, not the binary code, because students are not proficient enough in binary and assembly code analysis at FBK, UEL and POLITO. This might make our conclusions not applicable when only the binary is available to attackers. To mitigate this threat, the replication at UGent is conducted on the binary.
Internal validity threats concern external factors that may affect the independent variable. Subjects are not aware of the experimental hypotheses. Subjects are not rewarded for the participation in the experiment and they are not evaluated on their performance in doing the experiment.

Construct validity threats concern the relationship between theory and observation. They are mainly due to how we measure the effectiveness of code splitting and the related performance overhead. We manually assess the successful completion of each task in order to measure SR. During the labs, the experimenters make sure that times are accurately marked in the time sheets upon start and completion of the attack task. Performance measurements are repeated 100 times so as to remove the effect of the possible random fluctuations of the measured execution time under slightly different conditions.
External validity concerns the generalisation of the findings. In our experiment we considered one program, SpaceGame. Results may not generalise to different programs. However, the chosen program has been written by third parties, completely unrelated with ASPIRE, so as to ensure that it is not crafted to provide optimal application conditions for the ASPIRE protection. Moreover, the chosen program is publicly available as an open source project in SourceForge. Hence, it can be regarded as a representative for this category of software. Further replications of the study on additional objects will corroborate the external validity of our findings.

7.7 Dates

The four replications of the code splitting experiment will be conducted at Fondazione Bruno Kessler, Ghent University, University of East London and Politecnico di Torino between Autumn, 2015 and Spring, 2016.

8 Industrial case studies

Goal	Analyze the ability of ASPIRE to prevent DRM attacks
Treatments	T0 = original code; T1 = multiple protections applied
RQ1	To what extent do ASPIRE protections prevent attacks against DRM?
RQ2	What ASPIRE protections are most effective in preventing attacks against DRM?
Subjects	3-4 hackers from the NAGRA tiger team
Objects	DemoPlayer (binary code)
Tasks	Violate specific DRM protection
Data	Report about the different reverse engineering and attack activities (e.g., data de-obfuscation; identifier renaming; control flow reconstruction; code understanding; decompilation) carried out by attackers
Design	Long running (30+ days) case study

Table 9: Nagravision case study

Tables 9, 10, 11 provide a schematic overview of the industrial case studies. The *goal* of these case studies is to evaluate the degree of protection offered by the ASPIRE techniques as a whole, considering those operating on the source code as well as those operating on the binary code. The entire ASPIRE tool chain is applied to the industrial case studies, so as to ensure maximum protection. The subjects involved in these case studies are professional hackers employed by the industrial partners of ASPIRE.

8.1 Research questions

The case studies aim at answering the following research questions:

Goal	Analyze the ability of ASPIRE to prevent attacks against license protections
Treatments	T0 = original code; T1 = multiple protections applied
RQ1	To what extent do ASPIRE protections prevent attacks against license management?
RQ2	What ASPIRE protections are most effective in preventing attacks against license management?
Subjects	3-4 hackers from the SFNT tiger team
Objects	Diamante (binary code)
Tasks	Forge valid license
Data	Report about the different reverse engineering and attack activities (e.g., data de-obfuscation; identifier renaming; control flow reconstruction; code understanding; decompilation) carried out by attackers
Design	Long running (30+ days) case study

Table 10: SafeNet case study

Goal	Analyze the ability of ASPIRE to prevent attacks against secure authentication
Treatments	T0 = original code; T1 = multiple protections applied
RQ1	To what extent do ASPIRE protections prevent attacks against secure authentication?
RQ2	What ASPIRE protections are most effective in preventing attacks against secure authentication?
Subjects	3-4 hackers from the GTO tiger team
Objects	OTP (binary code)
Tasks	Authenticate with no valid credentials available
Data	Report about the different reverse engineering and attack activities (e.g., data de-obfuscation; identifier renaming; control flow reconstruction; code understanding; decompilation) carried out by attackers
Design	Long running (30+ days) case study

Table 11: Gemalto case study

- **RQ1** To what extent do ASPIRE protections prevent attacks against DRM/license management/secure authentication?
- **RQ2** What ASPIRE protections are most effective in preventing attacks against DRM/license management/secure authentication?

These research questions deal with the effectiveness of the ASPIRE protections, when these are applied to the industrial case studies. We want to assess the capability of the ASPIRE protections to resist to a massive attack mounted by professional hackers during a long time period. Moreover, we want to assess the relative importance of different defence lines implemented by the various components in the ASPIRE tool chain, by analysing the activities carried out by the professional hackers to defeat each specific ASPIRE protection.

8.2 Objects

The objects of this experiment are programs provided by the industrial ASPIRE partners:

Object	C SLoC	H SLoC	Java SLoC	Cpp SLoC	Total
DemoPlayer	2,595	644	1,859	1,389	6,487
Diamante	53,065	6,748	819	-	58,283
OTP	284,319	44,152	7,892	2,694	338,103

Table 12: Size of case study objects (measured by `sloccount`), divided by file type.

DemoPlayer: Media player provided by Nagravision and requiring DRM protection

Diamante: License manager provided by SafeNet

OTP: One time password authentication server and client

Table 12 shows some size data about the involved objects (reported SLoC include any library that must be compiled with the application code). The tasks that hackers are asked to perform on these programs are respectively:

- **Nagravision:** Violate a specific DRM protection of DemoPlayer
- **SafeNet:** Forge a valid license key that is accepted by Diamante
- **Gemalto:** Authenticate on OTP without having any valid credential

8.3 Data

Professional hackers are asked to compile a *Final Attack Report*. The attack report will cover the following points in detail:

1. **Type of activities carried out during the attack**: detailed indications about the type of activities carried out to perform the attack and the proportion of time devoted to each activity. For instance, hackers may want to indicate the following activities: (1) data de-obfuscation; (2) identifier renaming; (3) control flow reconstruction; (4) code understanding; (5) decompilation; (6) execution inspection (e.g., via debugger); (7) execution modification (e.g., via debugger scripts). Hackers should provide such classification for each working day, not just for the whole attack session.
2. **Encountered obstacles**: detailed description of the obstacles encountered during the attack attempts. In particular, hackers should report any software protection that they think was put into place to prevent the attack and that actually represented a major obstacle for their work.
3. **Attack strategy**: description of the attack strategy and how it was adjusted whenever it proved ineffective. Hackers should describe the initial attempts and the decisions (if any) to change the strategy and to try alternative approaches.
4. **Return of the attack effort**: quantification of the attack effort, if possible economically, so as to provide an estimate of the kind of remuneration that would justify the amount of work done to carry out the attack.

8.4 Design

The design of this experiment is a long running case study, with loose control on the involved subjects and mostly qualitative data collected during the execution of the experiment.

8.5 Qualitative analysis

Qualitative analysis of the reports collected from hackers will be the basis to answer RQ2. Evidence about the activities performed and the obstacles encountered will be mapped to the ASPIRE protections that were most effective in blocking the attacks mounted by professional hackers.
For what concerns RQ1, the answer may be boolean, i.e., the attack was or was not successful. However, in case of a non-successful attack, there might still be some degree of exploitation that was achieved, such as leakage of sensitive information, denial of service, or any other attack that was not the direct goal of the case study task. For this reason RQ1 is formulated in terms of the *extent* to which the attack is prevented. Again, qualitative data analysis will be employed to answer this question.

8.6 Threats to validity

The main threats to the validity of the case studies belong to the conclusion, internal, construct and external validity threat categories.
Conclusion validity threats concern the relationship between treatment and outcome. We assume that unsuccessful attacks can be attributed to the ASPIRE protections and that the obstacles encountered during successful attacks can be also attributed to the ASPIRE protections, while we do not know what would have happened without the ASPIRE protections. To mitigate this threat, we collect extensive feedback from the professional hackers, in order to be able to support our conjectures with objective evidence collected in the field.
Internal validity threats concern external factors that may affect the independent variable. While subjects are professional hackers who are used to the kind of attack tasks requested to them during the study, there might be factors out of our control that affect their performance. Being a long running case study, the degree of control that we can have on the activities performed daily by the professional hackers is limited. We reduce this threat to validity by introducing a structured and systematic data collection procedure.
Construct validity threats concern the relationship between theory and observation. They are mainly due to how we measure the effectiveness of the ASPIRE protections. We manually assess the successful completion of the attack tasks to decide on the answer to RQ1. For RQ2, we perform a qualitative analysis of the reports to obtain evidence in support to our conjectures.
External validity concerns the generalisation of the findings. Being based on a set of three case studies, results may not generalise to different cases. On the other hand, the considered objects are industrial applications, which make them quite meaningful cases, and the protected assets span across a wide and meaningful range (i.e., DRM, licenses, authentication), so we expect some degree of generalisability to similar applications and to similar industrial contexts.

8.7 Dates

The industrial case studies will be conducted at Gemalto, Nagravision and SafeNet between Spring, 2016 and Autumn 2016.

List of abbreviations

ACM CCS Association for Computing Machinery Conference on Computer and Communications Security

ACTC ASPIRE Compiler Tool Chain

ADSS ASPIRE Decision Support System

AKB ASPIRE Knowledge Base

API Application Programming Interface

ARO Army Research Office

ASPIRE Advanced Software Protection: Integration, Research, and Exploitation

ASM ASPIRE Security Model

AT Attack Time

CC Cyclomatic Complexity

CFIM Control Flow Indirection Metric

CRUD Create, Read, Update, Delete

DOP Number of destination register operands

DoW Description of Work

DPL Dynamic Program Length

DL Description Logic

DRM Digital Rights Management

DST Number of destination operations

EDG Number of edges

ET Execution Time

E/R DB Entity-Relationship DataBase

EXE Number of executed instructions

GUI Graphical User Interface

ICSE International Conference on Software Engineering

IDA Pro For clarification: IDA is not an abbreviation

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

INS Number of instructions

IND Number of index edges

JDK Java Development Kit

JMP Number of computed jumped instructions

MCAS Monte Carlo-based Attack Simulation

NB Number of bytes

OTP One Time Password

OWL Web Ontology Language

PF Protection Fitness Function

PO Performance Overhead

PN Petri Net

PNML Petri Net Markup Language

PNDV Petri Nets with Discrete Values

PSAM PN-based Software Attack Model

RNC Residue Number Coding

SAPS Single Attack Process Simulation

SAMMPN Software Attack Model based on Marked Petri Net

SLoC Source Lines Of Code

SOP Number of source register operands

SPRO Software Protection (Workshop)

SPA Software Protection Assessment

SR Success Rate

SRC Number of source operations

UML Unified Modeling Language

WP Work Package

XOR Exclusive OR

XML eXtensible Markup Language

References

[1] Gregory W Corder and Dale I Foreman. *Nonparametric statistics: A step-by-step approach*. John Wiley & Sons, 2014.

[2] Jay L Devore and Kenneth N Berk. *Modern mathematical statistics with applications*. Cengage Learning, 2007.

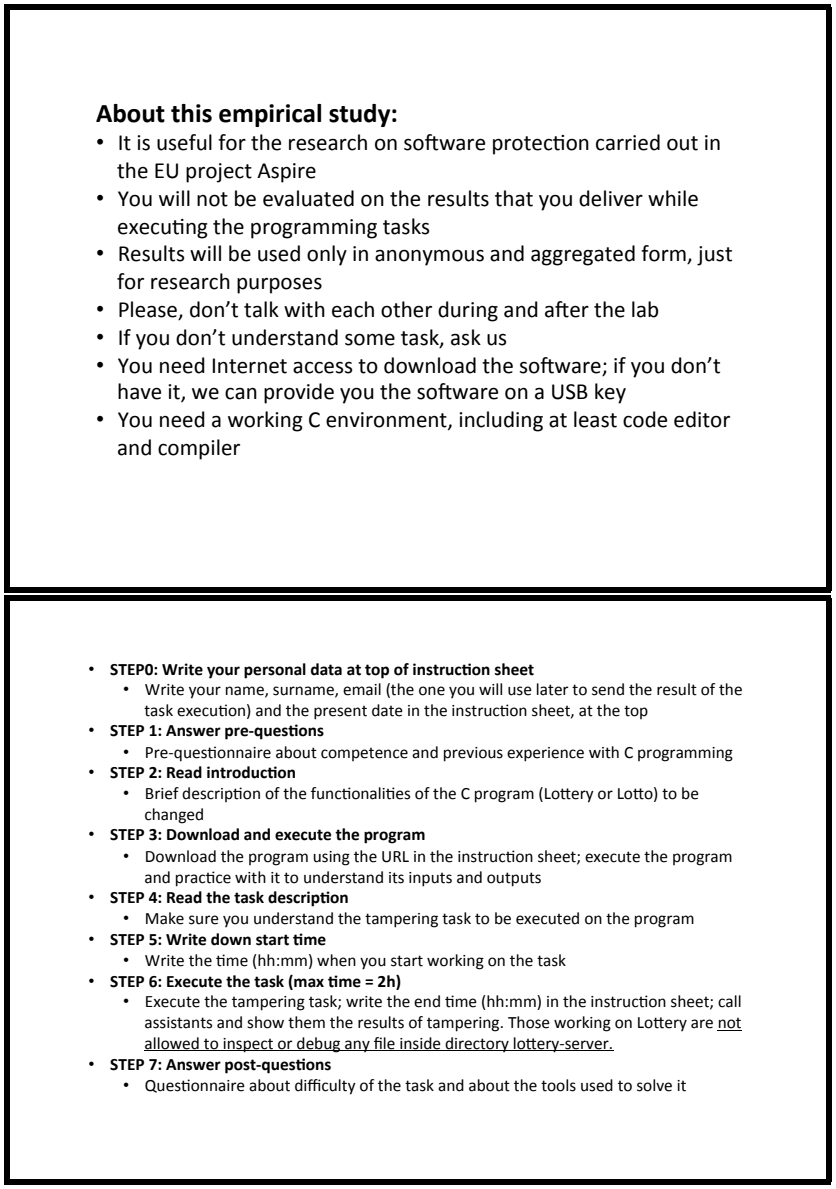
[3] K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[4] H. Wang, D. Fang, N. Wang, Z. Tang, F. Chen, and Y. Gu. Method to Evaluate Software Protection Based on Attack Modeling. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC*, pages 837–844. IEEE, 2013.

[5] David A Wheeler. More than a gigabuck: Estimating gnu/linux's size, 2001.

[6] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

A Introduction slides for the data obfuscation experiment on binary code

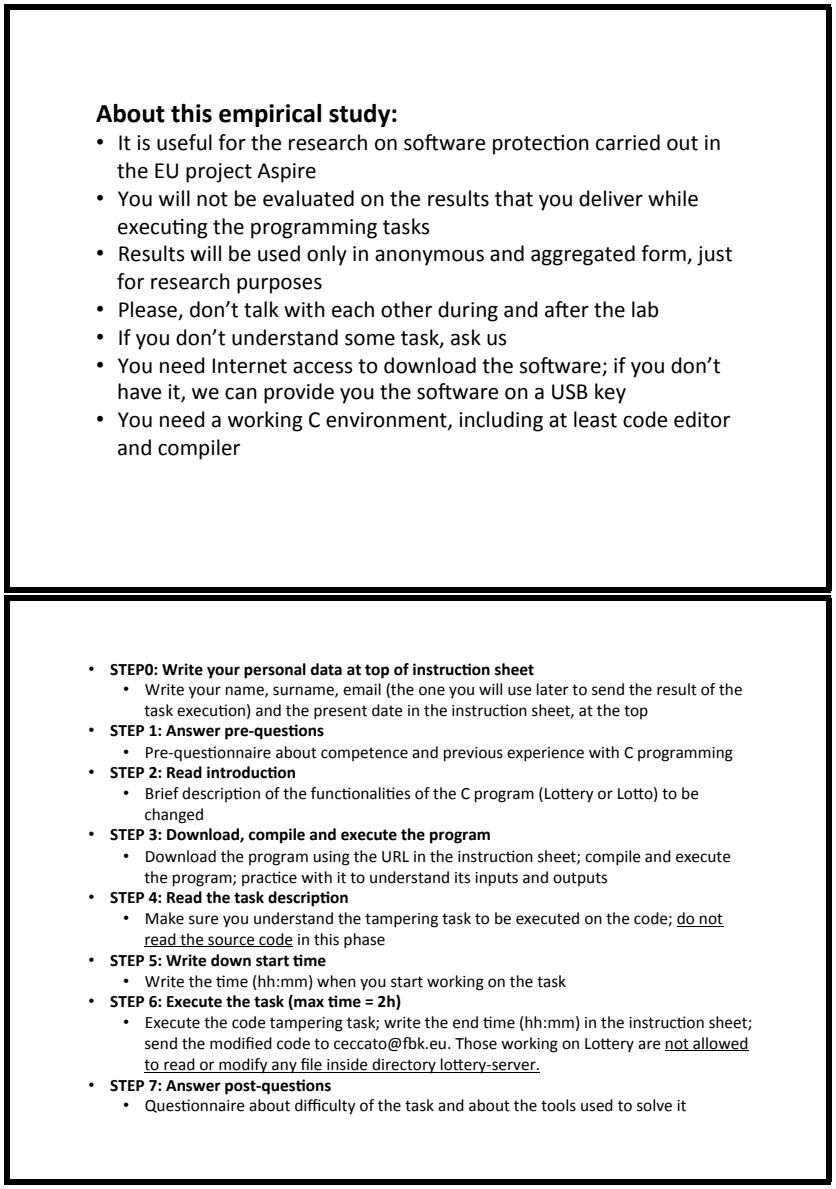


About this empirical study:

- It is useful for the research on software protection carried out in the EU project Aspire
- You will not be evaluated on the results that you deliver while executing the programming tasks
- Results will be used only in anonymous and aggregated form, just for research purposes
- Please, don't talk with each other during and after the lab
- If you don't understand some task, ask us
- You need Internet access to download the software; if you don't have it, we can provide you the software on a USB key
- You need a working C environment, including at least code editor and compiler

- **STEP0: Write your personal data at top of instruction sheet**
  - Write your name, surname, email (the one you will use later to send the result of the task execution) and the present date in the instruction sheet, at the top
- **STEP 1: Answer pre-questions**
  - Pre-questionnaire about competence and previous experience with C programming
- **STEP 2: Read introduction**
  - Brief description of the functionalities of the C program (Lottery or Lotto) to be changed
- **STEP 3: Download and execute the program**
  - Download the program using the URL in the instruction sheet; execute the program and practice with it to understand its inputs and outputs
- **STEP 4: Read the task description**
  - Make sure you understand the tampering task to be executed on the program
- **STEP 5: Write down start time**
  - Write the time (hh:mm) when you start working on the task
- **STEP 6: Execute the task (max time = 2h)**
  - Execute the tampering task; write the end time (hh:mm) in the instruction sheet; call assistants and show them the results of tampering. Those working on Lottery are not allowed to inspect or debug any file inside directory lottery-server.
- **STEP 7: Answer post-questions**
  - Questionnaire about difficulty of the task and about the tools used to solve it

B Introduction slides for the data obfuscation experiment on C source code



About this empirical study:

- It is useful for the research on software protection carried out in the EU project Aspire
- You will not be evaluated on the results that you deliver while executing the programming tasks
- Results will be used only in anonymous and aggregated form, just for research purposes
- Please, don't talk with each other during and after the lab
- If you don't understand some task, ask us
- You need Internet access to download the software; if you don't have it, we can provide you the software on a USB key
- You need a working C environment, including at least code editor and compiler

- **STEP0: Write your personal data at top of instruction sheet**
  - Write your name, surname, email (the one you will use later to send the result of the task execution) and the present date in the instruction sheet, at the top
- **STEP 1: Answer pre-questions**
  - Pre-questionnaire about competence and previous experience with C programming
- **STEP 2: Read introduction**
  - Brief description of the functionalities of the C program (Lottery or Lotto) to be changed
- **STEP 3: Download, compile and execute the program**
  - Download the program using the URL in the instruction sheet; compile and execute the program; practice with it to understand its inputs and outputs
- **STEP 4: Read the task description**
  - Make sure you understand the tampering task to be executed on the code; do not read the source code in this phase
- **STEP 5: Write down start time**
  - Write the time (hh:mm) when you start working on the task
- **STEP 6: Execute the task (max time = 2h)**
  - Execute the code tampering task; write the end time (hh:mm) in the instruction sheet; send the modified code to ceccato@fbk.eu. Those working on Lottery are not allowed to read or modify any file inside directory lottery-server.
- **STEP 7: Answer post-questions**
  - Questionnaire about difficulty of the task and about the tools used to solve it

C Instructions for the data obfuscation experiment on binary code

Group: C/T
Surname: **Name:**
Date: **Email:**

Lotto

STEP 1: Please, answer the following questions:

1. Have you ever worked as a professional programmer (in industry or in a computer house)?
   ○ No ○ Yes, part-time ○ Yes, full-time

2. What is your cumulative programming experience in C?

< 3 Months	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

3. How long have you been using an IDE for C programming?

Never	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

4. Can you do the following actions on a C debugger (multiple answers allowed)?
   ❑ Add breakpoints
   ❑ Execute the program stepwise
   ❑ Inspect the call stack
   ❑ Inspect the program variables

5. What is your cumulative programming experience in assembly?

< 3 Months	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

6. What is your cumulative programming experience with reverse engineering?

< 3 Months	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

7. Can you do the following actions on an assembly-level debugger (multiple answers allowed)?
   ❑ Add breakpoints
   ❑ Execute the program stepwise
   ❑ Inspect the call stack
   ❑ Inspect the program variables

STEP2: Please, read the following brief introduction

Lotto is a C program for machines that let users play lotto. A new program is generated every week, with the winning sequence embedded in the source code. The JACKPOT is hit when all 7 numbers are matched (i.e., all six numbers plus the bonus number).

STEP3: Compile, execute and practice

- Download the compiled code from:
  - `<UGENT-URL>/Lotto-C0234/T1298.zip`
- Execute Lotto, e.g., by running the command `./lotto`
- Practice with Lotto, playing Lotto for a few minutes, to get an understanding of how it can be played by users

STEP4: Please, read the following description of the task you will execute at STEP6
Determine the JACKPOT sequence, consisting of 7 winning numbers, which is embedded in the C source code (specifically, inside some variables) of the Lotto program. While executing the task, you are allowed to debug and execute the program. Write down the winning sequence below (under STEP 6).

STEP5: Write down the start time
Start time: _____

STEP6: Execute the task described at STEP 4, write the stop time and the winning sequence
Stop time: _____ Winning sequence: _____

STEP 7: Please, answer the following questions:
1. The task was clear to me.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

2. There was enough time to perform the task.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

3. The task was easy to perform.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

4. I have used the following tools to perform the task (multiple answers allowed):

Disassembler	IDE	Debugger	Internet search	Other: _____
❑	❑	❑	❑	❑

5. I have spent most of the time:

Reading and understanding the binary	Inspecting the execution by means of the debugger	Changing the execution by means of the debugger
O	O	O

Group: C/T
Surname: **Name:**
Date: **Email:**

Lottery

STEP 1: Please, answer the following questions:

1. Have you ever worked as a professional programmer (in industry or in a computer house)?
   ○ No ○ Yes, part-time ○ Yes, full-time

2. What is your cumulative programming experience in C?

< 3 Months	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

3. How long have you been using an IDE for C programming?

Never	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

4. Can you do the following actions on a C debugger (multiple answers allowed)?
   ❑ Add breakpoints
   ❑ Execute the program stepwise
   ❑ Inspect the call stack
   ❑ Inspect the program variables

5. What is your cumulative programming experience in assembly?

< 3 Months	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

6. What is your cumulative programming experience with reverse engineering?

< 3 Months	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

7. Can you do the following actions on an assembly-level debugger (multiple answers allowed)?
   ❑ Add breakpoints
   ❑ Execute the program stepwise
   ❑ Inspect the call stack
   ❑ Inspect the program variables

STEP2: Please, read the following brief introduction

Lottery is a client-server C program for machines that let users play lottery. Lottery uses a random sequence of bytes (called a *challenge*) generated by a lottery server to extract 7 numbers (without repetitions) between 1 and 39. The challenge is stored on the server, where it is used for anti-tampering check. A legal extraction consists of 7 numbers that match one of the challenges stored on the server. 10 extractions are performed by Lottery. The extracted numbers are printed by the lottery server into its log file (`log.txt`).

STEP3: Compile, execute and practice

- Download the compiled code from:
  - `<UGENT-URL>/Lottery-C5567/T8943.zip`

- Execute Lottery, e.g., by running the commands:
  - `./lottery-server > log.txt &`
  - `./lottery`
- Look at the output produced by Lottery (file `log.txt`), to get a sense of the reported statistics

STEP4: Please, read the following description of the task you will execute at STEP6
Using the debugger (either manually or through scripts), modify the execution of the program Lottery so that it extracts only numbers between 1 and 20 in a legal extraction (i.e., one matching the logged challenge). When any of the 7 extracted numbers is greater than 20, the extraction is redone, by requesting a new challenge to the lottery server. In fact, the extracted numbers are checked for validity against the challenge stored on the server. After successfully completing the task, the logged numbers shall be all less than or equal to 20. You are allowed to inspect and modify only program lottery; **the lottery server must not be inspected or debugged in any way**.

STEP5: Write down the start time
Start time: _____

STEP6: Execute the task described at STEP4, mark the stop time and show the log file to the assistants running the experiment
Stop time: _____

STEP 7: Please, answer the following questions:
1. The task was clear to me.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

2. There was enough time to perform the task.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

3. The task was easy to perform.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

4. I have used the following tools to perform the task (multiple answers allowed):

Disassembler	IDE	Debugger	Internet search	Other: _____
❑	❑	❑	❑	❑

5. I have spent most of the time:

Reading and understanding the binary	Inspecting the execution by means of the debugger	Changing the execution by means of the debugger
O	O	O

D Instructions for the data obfuscation experiment on C source code

Group: C/T
Surname: **Name:**
Date: **Email:**

Lotto

STEP 1: Please, answer the following questions:

1. Have you ever worked as a professional programmer (in industry or in a computer house)?
   ○ No ○ Yes, part-time ○ Yes, full-time

2. What is your cumulative programming experience in C?

< 3 Months	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

3. How long have you been using an IDE for C programming?

Never	6 Months	1 Year	2 Years	> 3 Years
○	○	○	○	○

4. Can you do the following actions on a C debugger (multiple answers allowed)?
   ❑ Add breakpoints
   ❑ Execute the program stepwise
   ❑ Inspect the call stack
   ❑ Inspect the program variables

STEP2: Please, read the following brief introduction

Lotto is a C program for machines that let users play lotto. A new program is generated every week, with the winning sequence embedded in the source code. The JACKPOT is hit when all 7 numbers are matched (i.e., all six numbers plus the bonus number).

STEP3: Compile, execute and practice

- Download the code from:
  - `http://selab.fbk.eu/asp/Lotto-C0234/T1298.zip`
- Compile Lotto, e.g., by entering directory src and running the command: `make`
- Execute Lotto, e.g., by running the command `./lotto`
- Practice with Lotto, playing Lotto for a few minutes, to get an understanding of how it can be played by users (**do NOT look at the source code in this phase!**)

STEP4: Please, read the following description of the task you will execute at STEP6

Determine the JACKPOT sequence, consisting of 7 winning numbers, which is embedded in the C source code (specifically, inside some variables) of the Lotto program. While executing the task, you are allowed to read, modify, compile, debug and execute the source code. Modify the program so that it prints the winning sequence to the standard output as soon as the program is launched.

STEP5: Write down the start time

Start time: _____

STEP6: Execute the task described at STEP4, mark the stop time and send the modified source code to ceccato@fbk.eu

Stop time: _____

STEP 7: Please, answer the following questions:

1. The task was clear to me.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

2. There was enough time to perform the task.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

3. The task was easy to perform.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

4. I have used the following tools to perform the task (multiple answers allowed):

Editor	IDE	Compiler	Debugger	Internet search
❑	❑	❑	❑	❑

5. I have spent most of the time:

Reading and understanding the code	Changing (e.g., adding printf) and executing the code	Executing the code with the debugger
O	O	O

Group: C/T
Surname: **Name:**
Date: **Email:**

Lottery

STEP 1: Please, answer the following questions:
1. Have you ever worked as a professional programmer (in industry or in a computer house)?
   O No O Yes, part-time O Yes, full-time

2. What is your cumulative programming experience in C?

< 3 Months	6 Months	1 Year	2 Years	> 3 Years
O	O	O	O	O

3. How long have you been using an IDE for C programming?

Never	6 Months	1 Year	2 Years	> 3 Years
O	O	O	O	O

4. Can you do the following actions on a C debugger (multiple answers allowed)?
   - ❑ Add breakpoints
   - ❑ Execute the program stepwise
   - ❑ Inspect the call stack
   - ❑ Inspect the program variables

STEP2: Please, read the following brief introduction
Lottery is a client-server C program for machines that let users play lottery. Lottery uses a random sequence of bytes (called a *challenge*) generated by a lottery server to extract 7 numbers (without repetitions) between 1 and 39. The challenge is stored on the server, where it is used for anti-tampering check. A legal extraction consists of 7 numbers that match one of the challenges stored on the server. 10 extractions are performed by Lottery. The extracted numbers are printed by the lottery server into its log file (`log.txt`).

STEP3: Compile, execute and practice
- Download the code from:
  - `http://selab.fbk.eu/asp/Lottery-C5567/T8943.zip`
- Compile Lottery, e.g., by entering directory src and running the command `make` and by entering directory src/lottery-server and running the command `make`
- Execute Lottery, e.g., by running the commands:
  - `./lottery-server/lottery-server > log.txt &`
  - `./lottery`
- Look at the output produced by Lottery (file `log.txt`), to get a sense of the reported statistics (**do NOT look at the source code in this phase!**)

STEP4: Please, read the following description of the task you will execute at STEP6
Modify the program Lottery so that it extracts only numbers between 1 and 20 in a legal extraction (i.e., one matching the logged challenge). When any of the 7 extracted numbers is greater than 20, the extraction is redone, by requesting a new challenge to the lottery server. In fact, the extracted numbers are checked for validity against the challenge stored on the server. After successfully completing the task, the logged numbers shall be all less than or equal to 20. Source code files

under src/lottery-server must not be read or modified during the execution of the task. In any case, **you are allowed to read and modify only file lottery.c.**

STEP5: Write down the start time
Start time: _____

STEP6: Execute the task described at STEP4, mark the stop time and send the modified source code (lottery.c) to ceccato@fbk.eu
Stop time: _____

STEP 7: Please, answer the following questions:

1. The task was clear to me.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

2. There was enough time to perform the task.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

3. The task was easy to perform.

Strongly agree	Agree	Not certain	Disagree	Strongly disagree
O	O	O	O	O

4. I have used the following tools to perform the task (multiple answers allowed):

Editor	IDE	Compiler	Debugger	Internet search
❑	❑	❑	❑	❑

5. I have spent most of the time:

Reading and understanding the code	Changing (e.g., adding printf) and executing the code	Executing the code with the debugger
O	O	O